# Assertion-Checker Synthesis for Hardware Verification, In-Circuit Debugging and On-Line Monitoring

**Marc Boulé**

Department of Electrical and Computer Engineering
McGill University, Montréal

A Thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical Engineering

February, 2008

*C'est parmi les hommes de cette espèce qu'est née l'idée horrible et dangereuse que la vie humaine tout entière n'est peut-être qu'une méchante erreur, qu'une fausse couche violente et malheureuse de la Mère des générations, qu'une tentative sauvage et lugubrement avortée de la Nature. Mais c'est aussi parmi eux qu'est née cette autre idée, que l'homme n'est peut-être pas uniquement une bête à moitié raisonnable, mais un enfant des dieux destiné à l'immortalité.*

— Hermann Hesse, Le Loup des steppes

## Acknowledgments

First and foremost I would like to thank my advisor, Prof. Zeljko Zilic, for mentoring and guiding me during my post-graduate studies. Working for over seven years under the wing of someone requires a great relationship, and for this I am eternally grateful. Thank you for giving me the space to explore some of my ideas, even when they did not lead to success. Thank you for pointing me in the direction of assertion-based verification, when all seemed lost. Thank you also for all the help reviewing my work and with publications and conferences.

Thanks to Derrick Wong from the Office of Technology Transfer at McGill University for all the work related to licensing and industry contacts, and also for the many discussions on intellectual property, patents, and Star Trek (live long and prosper!). Thanks to Jean-Samuel Chenard for co-authoring the papers on debug enhancements and silicon debug and for providing much needed feedback on the use of my checker generator. The Python scripts that you coded really helped make the experimental results in the thesis and papers less painful to produce. Thanks also to Stephan Bourduas, Jean-Samuel Chenard and Nathaniel Azuelos for being the first to use the checker generator in a real-world application.

I would also like to thank the many people who gave me advice at conferences. Special thanks are extended to Prof. Alan Hu, for suggesting relevant papers on automata and regular expressions in the early stages of my work. I also thank the members of the online verification community at `verificationguild.com`, and especially Cindy Eisner, who never spares any effort to help people with their questions about assertions. Seeing the quantity and quality of her responses, it is understandable that she would not waste time with capitalization!

On the financial side, I would like to thank my advisor for also providing me the necessary funds to help me pursue my doctoral studies.

J'aimerais remercier Patrick Cardinal d'avoir prêté l'oreille à toutes mes péripéties lors de mes recherches doctorales. Je le remercie aussi de l'aide apportée concernant la théorie des automates, d'avoir révisé nos articles, et de m'avoir prêté sa librairie d'automates, dont j'ai fait usage dans les premières versions de mon compilateur.

Je remercie André St-Amand de m'avoir initié à l'enseignement. Toutes ces séances de travaux dirigés en physique, en plus d'être fort intéressantes, m'ont apporté une variété intellectuelle qui n'a pu qu'être bénéfique. J'ai aussi beaucoup appris en te côtoyant dans l'aventure de ton livre sur la physique des ondes. Je peux dire avec assurance que mon cheminement dans les études graduées est en partie grâce à toi. Merci encore d'avoir participé avec intérêt et passion au développement de mon programme d'échecs, qui a servi de tremplin à mon projet de recherche en maîtrise. La théorie du chaos enseigne qu'un petit changement peut entraîner de grandes conséquences, et ne pas t'avoir connu aurait probablement eu comme conséquence la non-existence de cette thèse.

Je remercie également Dominique Borrione et Katell Morin-Allory du Laboratoire TIMA-VDS en France, pour les échanges intéressants concernant les différentes approches quant à la compilation d'assertions en circuits. Merci à Katell Morin-Allory d'avoir entrepris le développement des preuves assistées par ordinateur portant sur les règles de réécriture.

Du côté financier, j'aimerais remercier le Regroupement Stratégique en Microélectronique du Québec (ReSMiQ) pour la bourse d'études. Je remercie également le Fonds québécois de la recherche sur la nature et les technologies, pour la bourse de doctorat en recherche.

Je remercie mes parents et mes deux frères, à qui je dédie ce document.

# Abstract

Producing error-free circuits is of paramount importance in the semiconductor industry. Assertions are becoming an indispensable means of verifying the correctness of increasingly complex digital designs. Assertions model the proper behavior of a design, and are expressed in a high level language based on temporal logic. In dynamic verification, simulation is used to exercise a circuit in order to assess its behavior. For large designs, simulation times are often prohibitively excessive, and designs are instead emulated in hardware. Because of their high-level temporal operators, assertion statements do not lend themselves directly to hardware implementations such as emulation.

This thesis introduces techniques and algorithms for generating resource-efficient circuit-level checkers from hardware assertion statements. These checkers are added to the source design, where they monitor the appropriate circuit signals to find faulty execution sequences. In this work, a finite automaton framework and a set of algorithms are developed and used extensively to create an intermediate representation of assertions. Implementing the large variety of temporal operators found in properties is also performed using specially designed rewrite rules.

Checkers are circuit-level implementations of assertions, and thus allow assertions to be used in hardware emulation and simulation acceleration. The checkers are not only beneficial in pre-fabrication functional verification, but can also be used for debugging fabricated silicon, at speed, where timing issues are most prevalent. Using checkers beyond verification and silicon debug is also explored, by proposing the use of assertions and a checker generator to automate the design of certain types of circuits. A variety of enhancements are also introduced to improve the debugging process with assertion checkers. These enhancements range from additional observability and metric-reporting features, to behavioral modifications to the checkers.

The tool developed in this work outperforms the best-known checker generator, namely the FoCs Property Checkers Generator from IBM. Important improvements compared to FoCs are demonstrated in terms of the circuit-size of checkers, the behavioral correctness of checkers and the number of operators supported.

## Abrégé

La production de circuits exempts d'erreurs est d'une importance capitale dans le domaine des semiconducteurs. Avec l'augmentation constante de la complexité des circuits numériques, la vérification matérielle basée sur les assertions devient indispensable. Les assertions modélisent le bon fonctionnement d'un circuit, et sont spécifiées à l'aide d'un langage faisant appel à la logique temporelle. En vérification dynamique, la simulation est utilisée afin d'analyser le comportement d'un circuit. Cependant, les temps de simulation deviennent trop longs pour de gros circuits et par conséquent, ces derniers sont souvent émulés de façon matérielle. Étant donné la présence d'opérateurs de logique temporelle de haut niveau, les assertions ne sont pas directement implantables de façon matérielle.

Cette thèse présente les méthodes et les algorithmes nécessaires pour générer des circuits vérificateurs efficaces à partir des assertions. Ces vérificateurs se branchent au circuit à tester afin d'y observer les signaux, permettant ainsi de déceler un mauvais fonctionnement. Dans cet ouvrage, une série d'algorithmes ainsi qu'un modèle basé sur les automates finis sont développés et utilisés comme représentation intermédiaire pour les assertions. L'implémentation du vaste éventail d'opérateurs se fait aussi grâce à des règles de réécriture.

En créant des circuits vérificateurs, les assertions peuvent dès lors être utilisés dans l'émulation matérielle et les accélérateurs de simulation. Les vérificateurs sont déjà fort utiles lors de la vérification préfabrication. Ces circuits peuvent aussi être utilisés lors de la vérification de circuits manufacturés où les problèmes de cadençage sont les plus réalistes. L'utilisation des vérificateurs est aussi applicable au-delà de la vérification et du déverminage post-fabrication, et peut servir pour la conception de circuits de haut niveau. Un ensemble d'extensions aux circuits vérificateurs est aussi développé afin d'améliorer le processus de déverminage. Ces extensions portent autant sur l'augmentation de l'observabilité que sur l'ajout de métriques supplémentaires fournies à l'utilisateur.

L'outil servant à générer les circuits vérificateurs développé dans ce travail devance l'outil le mieux connu dans ce domaine, soit le générateur de vérificateurs d'IBM nommé FoCs. Les résultats démontrent des améliorations importantes concernant la taille et le bon fonctionnement des vérificateurs matériels, ainsi que le nombre d'opérateurs supportés par l'outil.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# List of Definitions

# List of Acronyms

| | |
|---|---|
| ABD: | Assertion-Based Design |
| ABV: | Assertion-Based Verification |
| ASIC: | Application Specific Integrated Circuit |
| ATPG: | Automatic Test Pattern Generation |
| BIST: | Built-In Self Test |
| BNF: | Backus-Naur Form |
| CTL: | Computation Tree Logic |
| CPU: | Central Processing Unit |
| CUT: | Circuit Under Test |
| CUV: | Circuit Under Verification |
| DFA: | Deterministic Finite Automaton |
| DFF: | D-type Flip-Flop |
| EDA: | Electronic Design Automation |
| EOE: | End of Execution |
| FF: | Flip-Flop |
| FPGA: | Field Programmable Gate Array |
| HDL: | Hardware Description Language |
| IC: | Integrated Circuit |
| IEEE: | Institute of Electrical and Electronics Engineers |
| IP: | Intellectual Property |
| LTL: | Linear Temporal Logic |
| LUT: | Lookup Table |
| MBAC: | Marc Boulé's Assertion Compiler |
| NFA: | Nondeterministic Finite Automaton |
| NOC: | Network On Chip |
| PSL: | Property Specification Language |
| RE: | Regular Expression |

|  |  |
|---|---|
| RTL: | Register Transfer Level |
| SERE: | Sequential Extended Regular Expression |
| SOC: | System On Chip |
| SVA: | System Verilog Assertions |

# Chapter 1

# Introduction

This chapter is intended as a short overview and introduction to the thesis. Many terms and expressions are used loosely in this chapter, and are explained in more detail in the Background chapter.

## 1.1 Context and Motivation

Producing high-quality Integrated Circuits (ICs) is of paramount importance in the semiconductor industry. In some cases the importance is of an economic nature, as product recalls and loss of market shares are but some of the consequences of providing faulty circuits. In military and aerospace applications, the consequences can be more dire especially when human lives enter the equation. Examples of defects in ICs range from satellite and rocket malfunctions, to glitches and failures in consumer applications. One well-known case is the floating point error found on early models of the Pentium processor by Intel. This "bug", known as the FDIV bug was not a fabrication or manufacturing defect, but rather a functional error in the design of the division algorithm. Although the effect was barely perceptible, the flaw was enough to cause a large recall of processors, and most estimates place the total cost to Intel at nearly half a billion US dollars.

With such high prices to pay for producing faulty devices, the electronics industry is constantly seeking ways to improve quality. Quality is defined as follows in the ISO 8402 standard for quality management and quality assurance [114]:

> *The totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs.*

In the field of digital systems, the ability of a system to satisfy any stated or implied needs is compromised by the presence of faults. Examples of types of faults that may be present in digital circuits are: design faults, fabrication faults and faults that arise during usage [120]. A logic bug in a Boolean function is an example of a design fault, whereas stuck-at-value faults and short circuits are examples of fabrication faults. When a correct circuit subsequently becomes faulty under unexpected conditions or prolonged use, these are termed user faults.

Verification aims to ensure that a circuit design behaves according to its intended specification. Verification deals with functional errors in circuit designs, in contrast to testing which deals with the detection of fabrication errors in ICs. In today's complex hardware digital circuit designs, verification typically requires more effort than the actual design phase itself. Recently, hardware assertions have become an important addition to the majority of verification efforts in practice. Assertions are additional statements that are added to a design to specify its correct behavior. Assertions have been used in software for many decades, and only in the last decade have they made a widespread impact in hardware design. In hardware, assertions model the correct behavior of the design using properties specified in temporal logic.

Assertions can also be seen as a formal documentation language, free of the ambiguities inherent to English language specifications. In another view, assertions can be seen as an executable specification, or a computable specification, when interpreted by verification tools. Once the assertion language is learned, adding assertions to a design to perform verification requires low overhead since assertions are text-based commands. Furthermore, assertions can be added incrementally as needed or as time permits. Even a single assertion can help find design errors. Engineers seeking to produce quality designs should have a desire to use mechanisms that help find flaws in their designs, and assertions are such mechanisms. After all, if an engineer is so confident in his or her design, there should be no problem with writing assertions that won't fail...

Hardware assertions are typically processed by simulators in dynamic verification, or by model checkers and theorem provers in static verification. When large designs are to be simulated, they are often emulated in hardware where the implicitly parallel nature of digital circuits can be exploited for faster run-times. To allow assertions to be used in hardware, a *checker generator* is required to transform the assertions into circuit-level *checkers*. Assertions are written in high-level languages and are not suitable for direct implementation in circuit form. Generating resource-efficient assertion-checking circuits is of primary importance when assertion-based verification is to be used in hardware emulation, post-fabrication silicon debugging and on-line monitoring applications.

Assertions are often first viewed as a means of performing, enabling or facilitating the task of hardware verification, hence the term Assertion Based Verification (ABV). However, to emphasize that assertions should be adopted in the earliest stages of the design cycle, the term Assertion Based Design (ABD) was introduced to convey this philosophy [74]. Using assertions at the initial specification and design phases can have important benefits throughout the remainder of the design cycle. One example of this is when separate design and verification teams work together to fix errors. Having a formally written set of properties upon which both teams can interact will help shorten the overall verification cycle.

The method of specifying hardware designs has evolved over the years, ranging from low-level schematic capture to the behavioral text based design languages that are currently in use today. However, as predicted by Foster et. al., the properties used in assertion based design could actually become the essence of design and verification [74]:

> *The way design and verification has traditionally been performed is changing. In the future, we predict that design and verification will become property-based.*

Properties are emerging as a new way of doing things in verification, and have the potential for becoming the new way of doing things in design. With the emerging adoption of assertions in verification, and their expected evolution as hardware design mechanisms, is it no wonder the assertion-based revolution is often referred to as a *paradigm* [121].

In this thesis, the Property Specification Language (PSL) is studied and used as the standard hardware assertion language. The many similarities between PSL and SVA, such as sequences for example, allow most developments made for PSL checkers to also apply to creating SVA checkers, as reported in the book based on this thesis [34]. A few key advantages associated to PSL are listed below, and help motivate its consideration in this work:

- PSL was the first hardware assertion language to receive IEEE standardization (IEEE–1850) [111];
- With the use of many flavor macros, PSL offers a language independent means of specifying assertions, and can be used in VHDL, Verilog, SystemVerilog, GDL and SystemC designs;

- PSL incorporates many temporal operators found in formal verification and model checking, such as Linear Temporal Logic (LTL) and Computation Tree Logic (CTL), and is therefore well positioned to serve both the formal verification community and dynamic verification scenarios (ex: simulation).
- The PSL language is arguably the most complex and expressive of assertion languages.

The Verilog language is used in this work as the underlying language for expressing circuit designs, thus the Verilog flavor of PSL will be used throughout. Verilog is the most widely employed Hardware Description Language (HDL) in use today [55].

The concept of checker generators for PSL was pioneered by IBM and their FoCs tool [108]. The original assertion language that served as the foundation for PSL was actually developed by IBM and was called "Sugar". The FoCs tool was originally intended for simulation use; however, the circuit checkers that are produced can also be synthesized in hardware. As witnessed in the history of publications (in chronological order) [29, 26, 30, 32], the checker generator developed in this thesis has consistently been at the forefront of generating resource-efficient checkers for circuit implementations. In one example, a checker that is three orders of magnitude smaller in terms of code size was produced, compared to the FoCs tool [29].

Automatic generation of checkers from assertions is much more advantageous than designing checkers by hand. For one, a single line of PSL can sometimes imply hundreds of lines of HDL code. Maintaining HDL checker code is also not desirable, as the specification can sometimes change during the development. Furthermore, if complex checkers are coded by hand they will likely contain errors themselves, which will need to be debugged.

The checker generator developed in this research is particularly geared towards producing assertion checkers that consume the fewest circuit resources when implemented in hardware. Many design choices and optimizations are performed with the underlying goal of reducing the circuit sizes of the checkers. In this document, unless specified otherwise, the expression "the checker generator" refers to the checker generator developed in this work, called MBAC.

## 1.2 Goals and Contributions

The principal set of problems addressed in this thesis is presented below as challenges, to more closely relate to the actual way these interrogations arose throughout the various stages of the research. The solutions, or answers to these challenges constitute the contributions made in this thesis, and are briefly outlined thereafter.

1. How can PSL assertions be converted into resource-efficient hardware checker circuits? Can all temporal operators suitable for dynamic verification be supported? If so, how?
2. How can these checkers be enhanced and/or modified to improve the debugging process?
3. Can assertion and a checker generator be used beyond verification?

The answer to challenge 1 is contained in Chapters 4 and 5, where the automata framework and the algorithms for converting PSL into automata, and subsequently into circuits, are developed. Over twenty automata algorithms are developed, and over thirty rewrite rules are introduced, all of which are specifically tuned to offer the most appropriate run-time monitoring semantics for hardware assertion checkers. Although a few algorithms and rewrite rules follow from classical automata theory and the PSL specification respectively, all are implemented in the context of a new dual-layer symbol alphabet and an efficient symbolic encoding for edge labels. Other novel developments include particular optimizations for minimization, and the introduction of mixed levels of determinism in the automata.

The answer to challenge 2 is contained in Section 6.4, where a series of debugging enhancements to assertion checkers is elaborated. New findings such as assertion threading, assertion completion and activity monitors are introduced in the context of automata-based assertion checkers.

Challenge 3 is explored in Section 6.5, where checkers are extended to silicon debugging and on-line monitoring scenarios. It is also in that section where the idea of using assertions and a checker generator to automate certain types of circuit design is developed, with an example application in redundancy control. The concept of mixing automata with separate logic gates is introduced in Section 6.3 for implementing a more efficient form of the eventually! operator, when compared to the rewrite rule developed in Chapter 5. The results of all challenges are assessed empirically in Chapter 7.

In this work, a checker generator is devised with particular uses in verification, silicon debugging and on-line monitoring. The verification in question can range from hardware emulation and simulation acceleration, to software-only interpretation of assertions as used in traditional simulators, and can even extend into formal verification by allowing certain types of properties to be used in model checkers that do not support PSL. The techniques used in developing the checker generator can further be applied to areas as diverse as hardware-accelerated protein matching (Section 7.4.1) and network intrusion detection, where hardware implementations of regular expression matching can help improve performance.

Adding circuit-level assertion checkers to a design invariably affects timing and floor-planning issues. As expected, the integrated circuit must have the necessary silicon area to implement the checkers. When the total size of a set of checkers exceeds the remaining area, methods such as assertion grouping (Section 6.5.3) can be used to help manage partitions of checkers in reprogrammable logic. Careful design practices can also be employed to help minimize the impact of adding checkers to a design. The emphasis in this work is placed on the actual checkers that are generated, as opposed to studying their effect on the source design, to which they can (and should) be attached.

## 1.3 External Contributions and Collaborations

The author would like to highlight the contributions made by Jean-Samuel Chenard regarding three co-authored publications [26, 27, 28]. A very useful partitioning script was developed to automate the individual synthesis of checkers when grouped in a single Verilog module. Another script was developed to automatically extract and convert the synthesis results into LaTeX formatted tables. These scripts were used extensively in the publication for which they were developed [27], and also for the majority of the experimental results in Chapter 7. The ideas of assertion grouping and management of checkers in programmable cores was also contributed by Mr. Chenard. Assertion grouping was also mentioned previously [2], unbeknownst to us at that time. Figures 6.7 and 6.12 appearing in Chapter 6 were originally designed by Mr. Chenard.

Jean-Samuel also coded the CPU pipeline and helped work out the related assertion threading example in Section 6.4 (Example 6.4) [28]. The author would also like to mention the contributions made by this collaborator regarding the debug enhancements [26, 27]. Many discussions helped develop and organize the debug enhancements, and in particular, the ideas of monitoring completion, adding counters for assertions and the concept of logging signal dependencies were brought forth by Jean-Samuel.

The message tracing assertion (NOC_ASR) used in Section 7.2 was devised in a collaboration with Stephan Bourduas, Jean-Samuel Chenard and Nathaniel Azuelos [48], where assertion checkers are explored for debugging hierarchical ring Network-On-Chip (NoC) designs.

The first versions of the automaton-based checker generator were developed using an automata library used in speech recognition at the CRIM (Centre de recherche informatique de Montréal). More specifically, the author acknowledges Patrick Cardinal for providing the library and helping with interfacing issues, as well as many helpful discussions pertaining to automata.

A contribution indirectly comes from IBM when in 2004 a freely available version of their PSL/Sugar parser was downloaded. This parser was extensively modified, but did serve as the starting point for the PSL and Verilog parser used in the author's work. The parser is used in the front-end of the checker generator and is responsible for creating a data structure representing the syntax tree of the input assertions.

Katell Morin-Allory from the TIMA-VDS group in France has also begun to perform machine assisted proofs of the rewrite rules presented in Section 5.4. At the time of writing, a handful of rules have already been proven, and the proofs of the remaining rules are forthcoming. It is during these related exchanges that semantics problem with the `never` operator were observed, and independently confirmed in the PSL issues list [113].

Although their results are not contained in this thesis, the author would nonetheless like to acknowledge the graduates and undergraduates at McGill University that made use of the checker generator in their projects (supervised by Prof. Zilic):

- Nathan Kezar and Smaranda Grajdieru, *Graphical User Interface for MBAC*, ECSE 494 – Design Project Laboratory, McGill University, June 12, 2006;
- Alya Al-Dhaher, *Course Project – Assertion Test Generation*, ECSE 649 – VLSI Testing, McGill University, May 11, 2006;
- Alya Al-Dhaher, *Automated Test Generation for Automata-based Assertion-Checkers*, Non-thesis Masters Project, McGill University, August 16, 2006;
- Hansel Sosrosaputro and Shiraz Ahmad, *Automation of Assertion Signal Monitoring*, ECSE 494 – Engineering Design Laboratory, McGill University, April 16, 2007.

## 1.4 Overview of the Thesis

The thesis begins by a presentation of relevant background material in Chapter 2. Important terms such as *checkers*, *emulation*, *silicon debugging*, *on-line monitoring* and *assertion-based verification*, to name a few, are explained. This chapter also contains an introduction to classical automata theory and regular expressions, and can be very helpful prerequisites to the automata framework and the PSL regular expressions appearing elsewhere in the thesis. Chapter 2 also contains an introduction to the PSL language, where its syntax is formally presented. An informal explanation of PSL's semantics is also given, and was preferred instead of the formal semantics contained in Appendix B in the PSL specification [111].

An overview of related research is presented in Chapter 3. Both the automata-based and interconnected-modular approaches to checker generation are introduced in the context of related research on assertion checkers. The use of automata in formal verification, more specifically in model checking, is also surveyed. The existing uses of assertions in simulators, emulators and in silicon debugging are explored. Other related research is presented at the end of that chapter, where other assertion languages are also mentioned.

Chapters 4 and 5 introduce the core notions to generating assertion checkers in this thesis. Chapter 4 introduces the automata framework used to symbolically represent assertions. Other important functions that are not specific to assertion automata are developed, such as determinization and minimization. The conversion of automata to circuit-level checkers is also developed at the end of Chapter 4.

The automata implementation of all assertion operators is introduced in Chapter 5. The presentation starts with the lowest layer in the language structure, namely Boolean expressions, then proceeds gradually to the language's intermediate layers, consisting of temporal sequences and properties, and concludes with the top-level verification directives. These two chapters alone represent sufficient material to describe the checker generation process, from start to finish.

Enhanced features and uses of checkers are then introduced in Chapter 6. The enhancements range from a more efficient implementation of the eventually! operator, to various debugging enhancements related to the checkers. The enhancements can be categorized as either modifications to the behavior of checkers, or as added capabilities in the observability and the reporting of metrics. Chapter 6 also presents a view of how checkers can be used beyond verification by proposing their uses in post-fabrication silicon debugging, as permanent on-line monitors for in-field status assessment, and even for performing high-level circuit design.

The checker generator is evaluated empirically in Chapter 7, using a variety of real-world and synthetic assertions. Themes such as assertion grouping, the choice of a symbol alphabet and the debug enhancements are evaluated. The typical evaluation consists in synthesizing the checkers for FPGA technology, where the resource utilization of the checkers can then be compared. Comparisons to the FoCs tool from IBM are also performed, and show that the MBAC checker generator produces smaller and faster circuits that offer the correct assertion behavior while supporting all operators.

Many of the details contained in this thesis were also submitted in a US patent application in September 2007 [31].

# Chapter 2

# Background

This chapter begins by positioning the main theme of the work within the areas of verification, post fabrication silicon debugging and on-line monitoring. The verification landscape and the assertion paradigm are also introduced. The topic of generating assertion checking circuits is also explained contextually, and motivated within the mentioned areas. An overview of regular expresssions and automata theory is then given, and the chapter ends with a presentation of the PSL language.

## 2.1 Hardware Verification, Silicon Debugging and On-line Monitoring

In the field of digital circuit design, designing a circuit for a particular task is one step, but designing a *correct* circuit is a whole other endeavor. Correctness has a different meaning depending on the stage at which it is sought. In the early specification stage, *validation* seeks to ensure that the design features meet the user's requirements: "are we building the right product?" In the design stage, *verification* aims to ensure that the design meets its given specifications: "are we building the product right?" In the manufacturing stage, *testing* seeks to detect fabrication faults in integrated circuits.

The tasks of verification and testing are increasing in complexity at a higher rate than the complexity of the designs themselves. This is known as the verification gap in the industry, and is also referred to as the verification crisis. The costly penalties associated to the release of faulty hardware are compounded by increased competition and decreased time to market. The motivation for performing a thorough design verification has never been greater, and will only increase in the future.

Assertion-Based Verification (ABV) [74] is emerging as the predominant method-

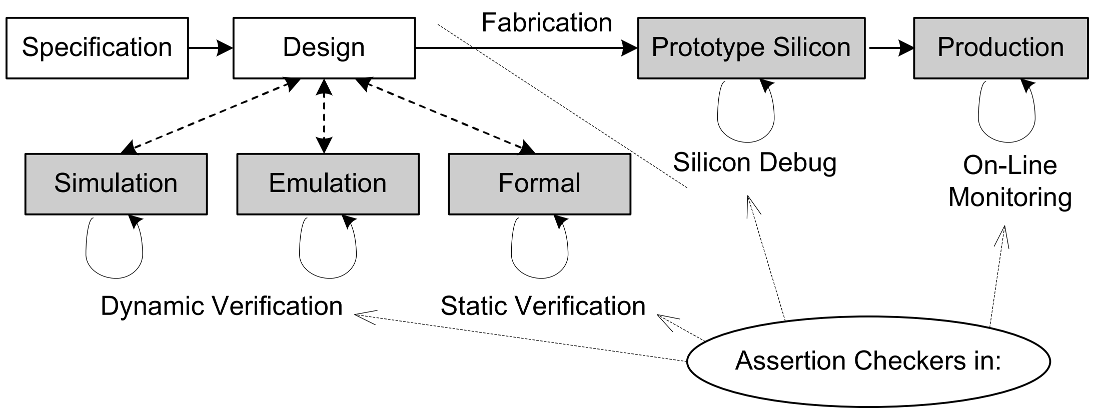


**Figure 2.1:** Assertion checkers in hardware verification, silicon debugging and on-line monitoring.

ology for performing hardware verification. Assertions are high-level statements built on temporal logic that are added to a circuit under verification in order to specify how the circuit should behave. Assertions can (and should) be added before the verification step, and should be part of the design process as well. Assertions can also be used the specification stage to allow the formal documentation of requirements. Figure 2.1 shows a summary of the main engineering tasks leading to a finished integrated circuit.

Assertions should ideally be adopted in the first two blocs in Figure 2.1, namely specification and design. When this is not the case, assertions are prepared during the verification steps for all three types of verification shown. Synthesizing assertion-checking circuits is an effective way of allowing assertions to be used in the verification, silicon debugging and on-line monitoring steps in the flow in Figure 2.1. The use of checkers in these various applications is overviewed throughout this section, and the presentation of checkers and checker generation (checker synthesis) is covered in Section 2.2.

Assertions have been used in software for many decades. In 1966 Robert Floyd is credited with formalizing assertions and launching the modern use of assertions when he developed the reasoning of programs based on axioms and predicate calculus. The history of computing, and more specifically the history of reasoning about programs [116] reveals that the concept of assertions was introduced in 1947 by Herman Heine Goldstine and John von Neumann, where the idea of assertion boxes for capturing the correct effects of programs were introduced. Two years later, in 1949, a paper by Alan Turing mentions adding assertions to a program (reported in [116]):

*How can one check a routine in the sense of making sure that it is right?*

*In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.*

Turing's citation above is surprisingly topical even for digital circuits, and by changing only a few words it can apply to today's hardware designs as well.

Assertions have only recently begun to be used in the field of hardware design. In fact, in 2004 the number of hardware engineers using assertions passed the 50 % mark (2004 DVCon census [54]). In 2007, the DVCon verification census [55] showed that 68.5 % of engineers were using assertions. It is also widely accepted that between 40 and 70 % of the design effort is actually spent on verification and bug-fixing [178]. With the increasing complexity of integrated circuits and hardware designs, this relatively high number is not surprising, and is not expected to diminish. Design teams are also increasingly being complemented by separate verification teams in large projects, where assertions play a key role.

Assertions are also very helpful in post-silicon debugging, where fabricated ICs (Integrated Circuits) are debugged under realistic timing conditions. In silicon debugging with assertions, assertion checkers can be purposely left in the fabricated IC for debugging purposes. Metrics from Intel show that in 2005, over 20 % of total design resources are spent on post-fabrication validation [146]. It is also reported that approximately 70 % of System-on-Chip (SoC) re-spins in the industry are due to logic bugs.

Various patterns are recurrent in verification [64], whereas others can be customized to a particular application. Assertions are a way of formally capturing the correctness properties (or patterns) of a specification. Hardware assertions are typically written in a Hardware Verification Language (HVL) such as PSL (Property Specification Language [111]) or SVA (SystemVerilog Assertions, part of the SystemVerilog language [110]). PSL is standardized by the Institute of Electrical and Electronics Engineers (IEEE) as the IEEE-1850 standard, and SystemVerilog is standardized as IEEE-1800.

In verification, assertions are the "golden rules" to which the implementation is compared. Any deviations from these golden rules constitute design errors. Assertions are not only beneficial in the verification process, but also represent an unambiguous way to document a design. Assertions have a formally defined syntax and semantics, and do not suffer from the inherent ambiguities commonly found in English language specifications. Formally documenting a design's specification with assertions can be seen as creating an executable specification [111] that can be automatically used by tools to assist in verification.

For example, the following statement expresses a certain specification that is to be respected by a bus arbiter (the reset signal is active low).

> *When the request signal goes from low to high in two consecutive cycles, then the grant signal must be asserted in at most five cycles and the request signal must remain high until this grant is received, unless a reset occurs.*

The same requirement can be expressed much more succinctly using an assertion language:

$$\textsf{assert always } (\{\sim req\ ;\ req\}\ |\!=\!>\ \{req[^*0{:}4]\ ;\ gnt\})\ \textsf{abort} \sim reset \tag{2.1}$$

In fact, the assertion above is actually more precise than the previous textual statement. Depending on the reader's point of view, the expression "*unless a reset occurs*" either only releases the obligation for the request signal remaining high until the grant is received, or it also releases the condition that the grant signal must be asserted with five clock cycles. In other words, it is not clear what portion of the statement the expression "*unless a reset occurs*" applies to. Furthermore, to what cycle is the expression "*in at most five cycles*" related? Is it the second or the first of the two consecutive cycles that are mentioned? Notwithstanding these ambiguities that are alleviated by using the formal assertion language, the assertion is also a much more compact form of documentation and specification.

The assertion shown in the example above was written in PSL, which will be covered in Section 2.4. The |=> operator is a temporal implication, the [**low*:*high*] operator is a form of repetition with a range, and the semicolon represents temporal concatenation. Assertions are typically bound to a design to be verified, which is called the *source design*.

The ABV methodology is based on the fact that the observation of an assertion failure helps to identify design errors, which are then used as a starting point for the debugging process. The amount of assertions that should be added to the design depends on the amount of coverage desired. One question that arises often with new ABV practitioners is: How many assertions do I need to write? The answer is not an easy one. Thankfully, assertions can be added incrementally, one at a time. Even a single assertion can help, and is better than having no assertions at all. However, in an ABV scenario, the fact that no assertions failed is not an indication that the design is entirely free of errors. It is more an indication that the behaviors specified by the set of assertions are respected.

The two main classes of functional verification are *dynamic verification* and *static verification*, and are visible in Figure 2.1. Static verification is typically associated with formal verification [11, 120], and is performed by tools such as automated theorem provers and model checkers. The term *static* is indicative of the fact that the circuit under verification does not need to be exercised. Model checkers and theorem provers analyze a model of the design, along with its properties or theorems, and are able to formally prove whether or not these properties or theorems are true.

One of the most popular model checkers is SMV [127], pioneered by McMillan. Many formal verification tools such as IBM's RuleBase and Cadence SMV, are built upon the SMV model checker. One particularly helpful feature in these tools is their ability to parse source designs in Verilog, against which formal properties can be verified. Example applications where RuleBase was used to perform formal property verification include: the verification of a processor bus interface unit [81] and a CoreConnect arbiter core [84]; the verification of bus interfaces such as PCI [7, 46] and FutureBus [51]; the verification of an MPEG2 decoder circuit [145], an OC-768 framer [63] and an SHA-1 hashing circuit [47].

Other formal verification systems are based on automated theorem provers, such as PVS [144] and ACL2 [119]. These systems complement their core theorem provers by adding programming languages and sophisticated proof commands. Among other applications, ACL2 was used to verify the floating-point arithmetic of the AMD K5 processor [157], and PVS was used to prove the correctness of a modular construction of checkers for PSL properties [135]. An embedding of PSL for the HOL theorem prover has been performed, and allows the formal reasoning of PSL properties for consistency checking [86, 87, 171].

Formal methods indicate a pass/fail result for the assertions, and in the event of a failure, counterexamples can also be generated. The advantage with formal methods is that stimuli do not need to be provided, and once a decision is reached, the result is proven correct. The disadvantage is that proving properties on complex designs can often be computationally expensive in time and/or processing requirements, or even impractical for large designs.

Dynamic verification is the predominant verification approach used in practice, and is most often associated with simulation. In dynamic verification, the design is exercised with a given stimulus, and its output is observed in order to assess the design's behavior. The disadvantage with dynamic verification is that stimuli must be provided, and covering hard-to-reach corner cases, let alone all cases, is difficult or impractical to achieve.

Assertions are also used in dynamic verification. Dynamic verification can be further categorized as pre-fabrication verification or post-fabrication debugging. Hardware verification and design verification are often used to refer to pre-fabrication verification, where simulation and emulation are the principal techniques employed. In verification by simulation, the simulator analyses the execution run and reports on the status of the assertions. The advantage with simulation verification is that it is easier to setup and does not require the more advanced technical and mathematical skills required to operate model checkers and theorem provers. A minimal skill set is nonetheless required to use assertions in simulation, as the temporal logics of assertion languages also have their share of mathematical and formal notions.

Assertions can also play an important role in post-fabrication silicon debugging, where assertion checkers are purposely left in the fabricated silicon for in-chip at-speed debugging. The relation between silicon debugging and design verification is shown in Figure 2.1, where the separation between the two is the fabrication step. Assertion-checking circuits can even be used for more than verification and debugging, and can also be incorporated into an IC to perform in-field on-line status monitoring (Figure 2.1). In this way a device can automatically assess its operating conditions, whereby the assertion checkers are used as a means of monitoring the device. The items in gray in Figure 2.1 show the scope of where the checkers developed in this work apply.

When assertions are interpreted by verification tools, a pass/fail result is the minimal feedback that a tool must provide. However, by reporting the clock cycle(s) where the assertions have failed, much more insight is gained to help determine the causes of the errors.

In dynamic verification with assertions, proper care should also be taken to build a testbench [13] that covers, as most as possible, a meaningful and relevant set of scenarios. If an assertion did not fail because of a lack of proper stimulus, this is not an indication that the design is error-free. Coverage is perhaps the main caveat with dynamic assertion-based verification, or any dynamic verification scenario for that matter.

Many simulators such as ModelSim [129] and Synopsys VCS can interpret assertions in order to use ABV in dynamic verification. However, as circuits become more complex, simulation time becomes a bottleneck in dynamic verification. Hardware emulation is becoming an important asset for verification, and is increasingly being used in the industry to alleviate the problem of excessive simulation time [41]. Hardware emulation achieves the traditional dynamic verification goals by loading and executing the design on reprogrammable hardware, typically using programmable logic devices or arrays of processing elements. Once the design is implemented in hardware, the emulator fully exploits the inherent circuit parallelism, as opposed to performing a serial computation in a simulation kernel.

Adding checkers to a design can allow assertions to be indirectly processed by simulators that do not support the assertion languages. The same can be done in formal verification, where checkers can be used to allow model checkers to indirectly support PSL or SVA assertions. In such cases, certain types of assertions can be checked by stipulating a temporal property stating that the checkers will not report any errors. The application of checkers in both of these forms of verification was also illustrated in Figure 2.1.

To summarize, there is a vast array of scenarios where assertions and assertion checkers play an important role: verification, hardware emulation, post-fabrication debugging, permanent online monitoring, simulation and formal verification. Synthesizing assertion checkers is beneficial and in most of these cases essential, to allow the assertion paradigm to be used in these areas. Assertion checkers, and checker generators are introduced next as a means of enabling assertion usage in this wide ranging and non exhaustive set of applications.

## 2.2 Assertion Checkers and Checker Generators

Assertion languages allow the specification of expressions that do not lend themselves directly to hardware implementations. Such languages allow complex temporal relations between signals to be stated in a compact and elegant form. To allow assertion-based verification to be used in hardware emulation, a *checker generator* is used to generate hardware assertion checkers from assertion statements [1, 30]. These checkers are typically expressed in a Hardware Description Language (HDL). A checker generator can be seen as a synthesizer of monitor circuits from assertions. An assertion *checker* (or assertion circuit) is a circuit that captures the behavior of a given assertion, and can be included in the design under verification for in-circuit assertion monitoring. The *assertion signal* is the result signal of a checker and is the signal that is monitored during execution to identify errors.

Figure 2.2 shows a high-level view of the assertion-based verification methodology, and the roles played by assertions, checkers and the checker generator. At the left

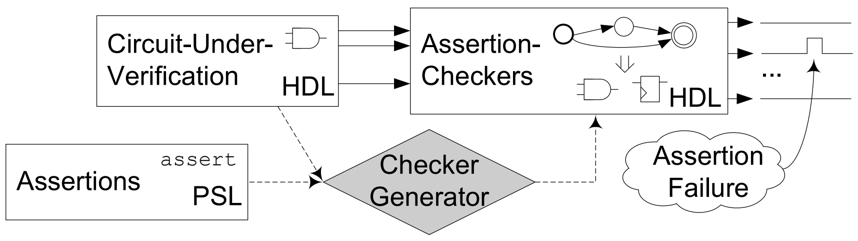


**Figure 2.2:** Checker generator for hardware verification.

of the figure are the given inputs to the tool, namely the Circuit Under Verification (CUV) and the assertions. In this example the circuit is described in a HDL and the assertions are specified in PSL. The checker generator produces an assertion circuit (a checker) for each input assertion. In this example, the checkers are transformed from an intermediary representation in automaton form. Automata theory facts will be explained further given that the checker generator developed in this work is automaton based.

The checker generator must also have access to the source design so that the signal dimensions can be known and the proper signals can be declared in the checkers. As is also shown in the figure, the checkers are connected to the CUV to monitor the proper signals. The output of a checker consists of a single signal, which is normally at logic-0, and becomes asserted (set to logic-1) in the clock cycles where the assertion fails. The checker generator developed here assumes that the designs are synchronous and that a clock signal is always present.

To give more insight into the task that must be performed by an assertion checker, the following property is analyzed both through the formal and the run-time verification paradigms. Boolean signals $a$ and $b$ are used, however without loss of generality more complex Boolean expressions can be used as well. As will be observed, the property is handled differently depending on the application; however, the conclusions reached are identical.

$$\mathsf{always}\ a\ \mathsf{->}\ \mathsf{next}\ b \tag{2.2}$$

The example property above makes use of property implication and the next operator. The property states that: it should always be the case that if signal $a$ is true in a given clock cycle, then signal $b$ must be true in the next cycle. The formal syntax of PSL and its semantics will be explained further in this chapter.

Figure 2.3 shows how the property is interpreted in both formal and run-time (dynamic) property checking. In formal verification, the property is checked against a model of the design under verification. This is usually represented by a state

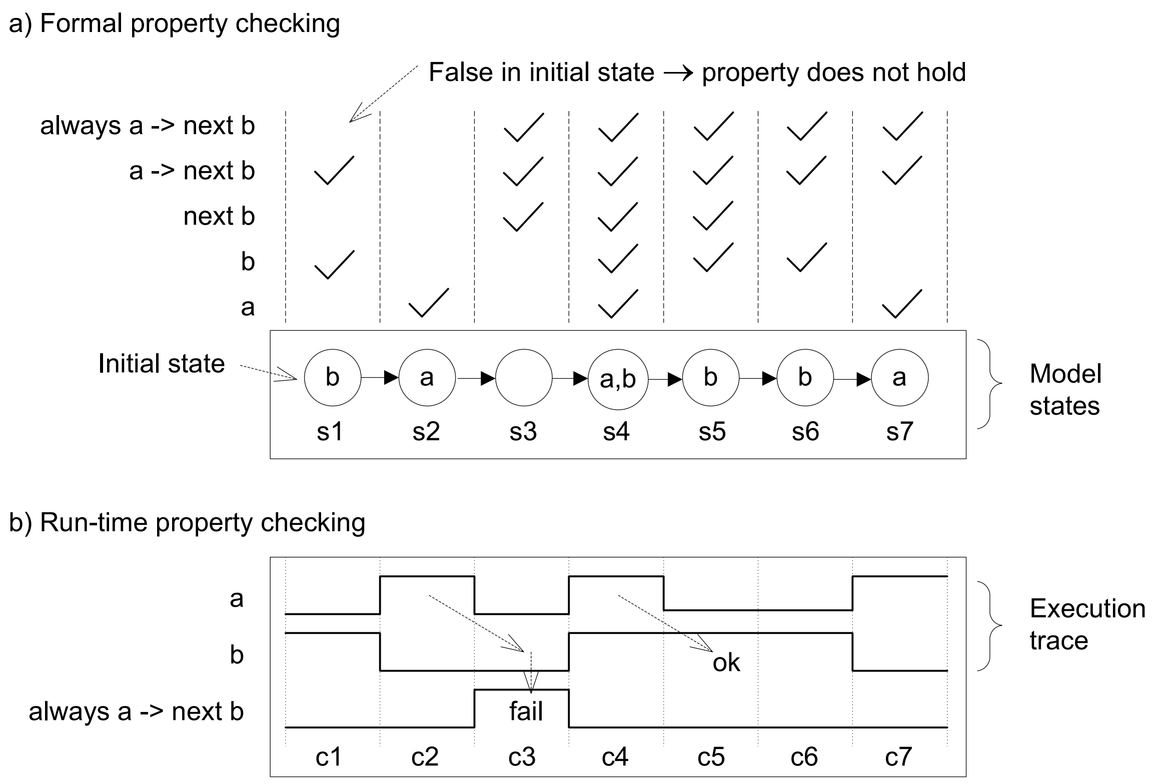


**Figure 2.3:** Formal vs. run-time property checking example.

transition graph, and in this example, in Figure 2.3 a) the values inscribed in the states represent the Boolean signals that evaluate to true in those states. This is a simplified example and in reality, the model can be much more complex and may contain branches and cycles. Furthermore, the property shown above employs Linear Temporal Logic (LTL); however, model checking with the branching time logics of CTL (Computation Tree Logic) is also possible. Branching time logics such as CTL are not suitable for the monotonically advancing time of dynamic verification, and will not be studied here.

In the top part of Figure 2.3 a), the property is decomposed into smaller constituents, and each sub-expression is evaluated in a bottom-up manner. The model checking starts by evaluating the terminals (the Boolean signals), and then gradually evaluates larger sub-expressions up to the full property. For example, $a$ is true in states 2, 4 and 7, and this is represented by a check mark. Boolean $b$ is true in state 4, thus next $b$ will be true in state 3, and so forth. The implication $a$ -> next $b$ is true in any state where $a$ is not true (1, 3, 5, 6, 7), and also in state 4 where both $a$ and next $b$ are true. The sub-property $a$ -> next $b$ does not hold in state 2.

In a given state, the always operator evaluates to true when its argument property is true starting in that state and for the remaining states. In this example, state 7 is a terminal state because it has no outgoing transitions. The always property is true in states 3 to 7 because from any of those states, $a$ –> next $b$ is true in all the remaining states. The top-level property fails in states 1 and 2 because the sub-property fails in state 2. Since the property fails in the initial state (state 1), the property does not hold. This is the pass/fail answer (i.e. fail), and two counterexamples can be deduced by observing the sequences starting in state 1 and in state 2.

In Figure 2.3 b), the context is dynamic verification. In dynamic verification, the model of the design is not required, and an execution trace of the design is instead used. A *trace* is a waveform of signals showing their instantaneous values in the vertical axis, as a function of time (horizontal axis). The assertion must be checked against this trace, to determine if it passes or if it fails.

One solution for performing dynamic verification consists in storing the entire trace and building a model to represent the trace, and then using model checking to verify the property [88]. However, capturing an entire execution trace for long durations or for many signals is not always practical.

In run-time property checking, the assertion checkers must produce a decision in real-time, and the trace-storing solution is not applicable. For optimal debugging, the checkers should provide an output signal of their own, thereby indicating when the property fails. The convention used in the figure is that a property signal is normally low, and is asserted in cycles where the property fails. The example property is interpreted at run-time as shown at the bottom of the figure.

The key observation here is that unlike in static verification, in the run-time analysis the future value of a signal can not be known in the current cycle. In state 2 in Figure 2.3 a), it is known that $b$ is not true in the next state and that the property fails in state 2. However, in run-time verification, in clock cycle 2 (c2) the future value for $b$ has not occurred yet, and there is no failure to report. The failure can only be reported once cycle 3 has taken place and the checker observes that $b$ is not true. In the example, the run-time checker identified a failure in clock cycle 3.

As a side note, if the always operator was not used in the example property, both methods would report that the property holds, because $a$ –> next $b$ is true in the initial state. One of the goals in this work will be to develop the necessary algorithms to implement circuits that can perform run-time checking of assertions.

To conclude the example, both methods reach the same conclusion: the property does not hold. In model checking, counterexamples were produced, and in run-time property checking, the locations of faulty traces (also akin to counterexamples) were identified. It is precisely because of these differences in interpretation of properties that the PSL specification does not dictate the run-time semantics for the interpretation of PSL. This is best explained by the very insightful and evocative remark by Eisner [67]:

> *PSL defines whether or not a property holds on a trace – that is all. It says nothing about when a tool, dynamic or static, should report on the results of the analysis.* (2.3)

An important corollary of this fact is that in dynamic verification, two separate tools may produce different behaviors in the output traces of their assertion checkers, while both still being correct. This makes a direct comparison of checkers slightly more troublesome when a cycle-by-cycle comparison is attempted for checkers from two different tools.

The example assertion used in (2.2) makes use of the weak temporal operator `next`, which places no obligation for the next cycle (or state) to take place. In dynamic verification, if $a$ occurs in the last simulation cycle then the property may hold because the next cycle will never occur. If this is an unacceptable condition, the strong version of this operator could instead have been used (i.e. `next!`), thereby indicating that any antecedent condition has to see its next cycle realize or else the property fails.

For example, if the operator `next!` was used instead of the weak `next` in (2.2), the dynamic verification scenario in Figure 2.3 b) would contain an additional failure in clock cycle 7. Also, in the formal verification scenario there would be no check marks in the top row in Figure 2.3 a), and many more counterexamples could be reported.

The use of strong operators leads to the following interrogation: how does a run-time assertion checker know when the simulation (or execution) is finished? In each clock cycle a decision is taken and the status of the assertion is reported. Nothing stops the user from halting the simulation and running an additional number of steps in the future. In hardware assertion checking, the solution consists in providing a special end-of-simulation signal that is used to inform the checkers that time is considered finished and that no further cycles will be run. The checkers that require this signal utilize it to indicate additional failures when strong obligations are not met. This technique was developed in the FoCs checker generator [108], and is also employed in the MBAC checker generator.

Since in this work the checkers can be used directly in circuits or in emulation platforms, the end-of-simulation signal will be referred to as the *End Of Execution* (EOE) signal. It is assumed that when this signal is required, the checkers declare it

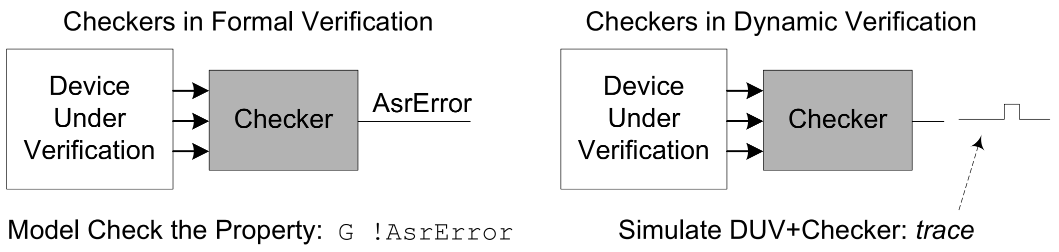

Checkers in Formal Verification
Device Under Verification
Checker
AsrError
Model Check the Property: G !AsrError
Checkers in Dynamic Verification
Device Under Verification
Checker
Simulate DUV+Checker: trace

**Figure 2.4:** Using checkers in formal and dynamic verification.

as an input, and the user normally drives it at logic-0, and raises it to logic-1 for at least one clock cycle at the end of execution.

The example studied in this section, shown in (2.2) and in Figure 2.3, was analyzed in both the static and dynamic verification approaches. When simulators or model checkers do not support PSL, generating assertion checkers and adding them to the source design is an effective way of allowing the continued use of assertions. In both cases the checkers are connected to the design under verification, as shown in Figure 2.4. In the simulation case, the output signal of the checker can be observed and any violation can be identified in the trace. In the formal verification case, a simple property is stated to postulate that the checker output(s) are always (G) false. This is an LTL property that is implicitly checked over all possible execution paths. Using checkers in formal verification is straightforward for safety-type properties (invariants), but would require some adaptation for liveness-type properties, which apply to infinite executions.

This thesis introduces the algorithms to convert PSL assertions into efficient checker circuits for use in hardware verification, in-circuit monitoring and post-fabrication silicon debugging. The checkers are particularly suited for hardware implementation where circuit speed and resource efficiency are paramount. Assertion circuits should be compact, fast and should interfere as little as possible with the design being verified, with which the checkers share the hardware resources.

## 2.3 Regular Expressions and Classical Automata

Assertion languages such as PSL and SVA make heavy use of sequences to specify temporal chains of events. Regular Expressions (REs) are the basis upon which PSL's Sequential-Extended Regular Expressions (SEREs) are built, and are an important preamble. Conventional automata are presented in this subsection as well, and will be used further as a base upon which to define the automata framework used for creating assertion checkers. Although many differences will arise, a comparison to conventional automata will help clarify the presentation of the automata developed for assertions. The themes in this section are based on the theory of automata, languages and computation [101].

A string is a sequence of symbols from an alphabet $\Sigma$, including an empty string, denoted as $\epsilon$. A regular expression $RE$ is a pattern that describes a set of strings, or a *language* of $RE$, denoted $L(RE)$. For example, if the alphabet consists of the ASCII characters, regular expressions efficiently represent a set of strings that can be searched for in string matching.

**Definition 2.1:** *Regular Expressions* (REs) and their corresponding languages are described as follows, where $r$ is a regular expression:

- a symbol $\alpha$ from $\Sigma$ is a RE; $L(\alpha) = \{\alpha\}$
- $\epsilon$ and $\emptyset$ are REs; $L(\epsilon) = \{\epsilon\}$ and $L(\emptyset) = \emptyset$
- $r_1|r_2$ is a RE; $L(r_1|r_2) = L(r_1) \cup L(r_2)$ (set union)
- $r_1 r_2$ is a RE; $L(r_1 r_2) = L(r_1)L(r_2)$ (set concatenation)
- $r_1*$ is a RE; $L(r_1*) = (L(r_1))*$ (Kleene closure)

The Kleene closure (Kleene star) is an operator that creates the strings formed by concatenating zero or more strings from a language. Parentheses can also be used for grouping, and as usual, $\emptyset$ denotes the empty set. It should be noted that $L(\epsilon) \neq L(\emptyset)$, as the former describes a non-empty language formed by the empty string, whereas the latter describes the empty language (called null language in this work).

A regular expression's language can be captured equivalently, in a form suitable for computation, by a finite automaton that accepts the same language.

**Definition 2.2:** A *classical Finite Automaton* (FA) is described by a quintuple $A = (Q, \Sigma, \delta, q_0, F)$ as follows:

- $Q$ is a non-empty set of states;
- $\Sigma$ is the alphabet;
- $\delta \subseteq Q \times \{\Sigma \cup \{\epsilon\}\} \times Q$ is the transition relation;
- $q_0$ is the initial state;
- $F \subseteq Q$ is the set of final states.

The non-empty set $Q$ is a finite set of locations (states). The alphabet $\Sigma$ is the same alphabet that was described above for regular expressions. A transition (or edge) is represented by an ordered triple $(s, \sigma, d)$, and the transition relation consists of a subset of triples:

$$\{(s, \sigma, d) \mid s \in Q,\ \sigma \in \{\Sigma \cup \{\epsilon\}\},\ d \in Q\}$$

The transition relation is sometimes expressed as the mapping $Q \times \{\Sigma \cup \{\epsilon\}\} \rightarrow 2^Q$. The transition relation indicates which destination state(s) to activate, for each state and for each input symbol that is received. The transition relation does not need to be complete and a state does not always have to activate other states.

Identifiers $s$ and $d$ refer to the source and destination states of an edge, respectively. An edge also carries a symbol $\sigma$ taken from the alphabet $\{\Sigma \cup \{\epsilon\}\}$. If an edge carries the $\epsilon$ symbol, then the state transition is instantaneous. When matching against an input string, no input symbol needs to be processed for an $\epsilon$ transition to take place. For a non-$\epsilon$ edge whose source state is active, a given transition takes place when the input symbol is identical to the edge's symbol.

The automaton has a single initial state. When the pattern matching begins, this is the only active state. A subset of states can also be designated as final states. When a final state is active, the pattern described by the automaton has been matched. Final states are also called accepting states, and they can be seen as accepting the language modeled by the automaton.

The automaton represents a pattern matching machine that detects all strings that conform to the language represented by the automaton. In other words: if $A$ is an automaton built from a regular expression $r$, then $L(r) = L(A)$. More generally, the input string in pattern matching is called a word, and an element of the word is called a letter. The alphabet's symbols are all mutually exclusive and one and only one letter is processed at a time by the automaton. At each step the automaton transitions into a new set of active states.

The convention used in this work is to represent the initial state using a bold circle and final states using double circles. A regular expression is converted into an equivalent automaton in a recursive manner, as shown in Figure 2.5. First, terminal automata are built for the symbols of the regular expression, as shown in part a). The empty automaton and the null automaton are acceptors for the languages $L(\epsilon)$ and $L(\emptyset)$ respectively, and are shown in parts b) and c).

Next, these terminal automata are inductively combined according to the operators comprising the given RE. The Kleene closure of an automaton is created by adding $\epsilon$ edges for bypassing the automaton (empty matching), and re-triggering the

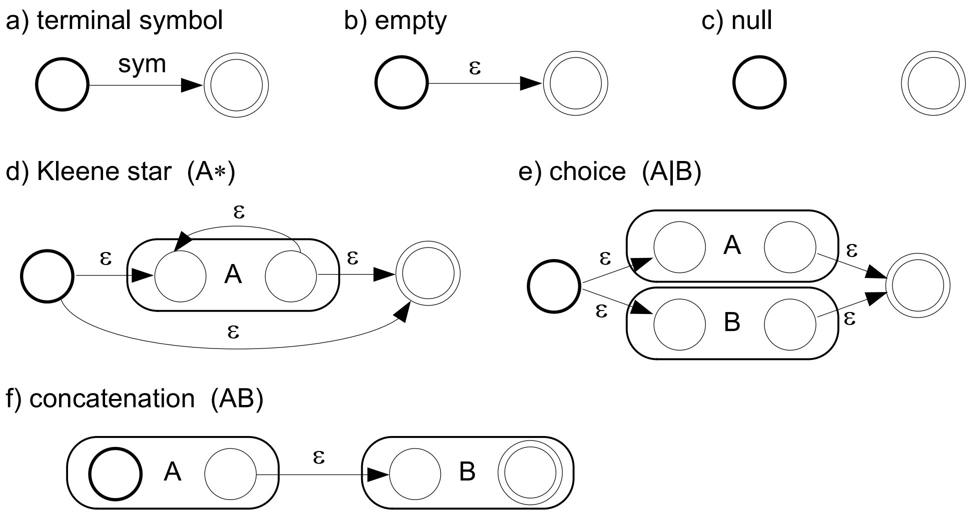


**Figure 2.5:** Automaton construction steps (McNaughton-Yamada construction).

automaton (multiple consecutive matches); this is illustrated in part d) of Figure 2.5. Choice and concatenation of two argument automata involve combining the automata using $\epsilon$ edges, as shown in parts e) and f).

The construction procedure described above, called the McNaughton-Yamada construction [128], produces a Nondeterministic Finite Automaton (NFA) containing $\epsilon$ transitions. An automaton can be determinized, hence producing a deterministic finite automaton (DFA).

**Definition 2.3:** An automaton with a single initial state and no $\epsilon$ transitions, and where no state can simultaneously transition into more than one successor state is a *Deterministic Finite Automaton* (DFA), otherwise it is a *Nondeterministic Finite Automaton* (NFA).

This definition is broad in nature and will apply equally to the automata introduced in Chapter 4. A corollary to Definition 2.3 can be made for classical automata, where the alphabet symbols are mutually exclusive entities that are received one at a time by the automaton, and thus no two symbols can occur at once.

**Corollary 2.4:** By extension from Definition 2.3, a classical FA is a Deterministic classical FA when it has a single initial state and no $\epsilon$ transitions, and when no more than one outgoing edge from a given state carries the same symbol, otherwise it is a *Nondeterministic classical Finite Automaton.*

The definition for DFAs (Definition 2.3) is consistent with Watson's work [177], but is different than what is proposed by Hopcroft *et. al.* [101]. For some authors, the definition of DFAs is such that every state must transition into *precisely one* successor state (as opposed to *at most one* successor state). When every state in a DFA transitions into precisely one next state, the DFA is said to be complete. In this work, determinization does not imply completeness and a separate definition is therefore used.

**Definition 2.5:** A DFA for which every state transitions into precisely one successor state at each step is a *complete DFA*.

The corollary of completeness in the case of classical automata follows naturally.

**Corollary 2.6:** By extension from Definition 2.5, a classical FA in which every state has one outgoing transition for every symbol is said to be a *complete classical DFA*.

The two corollaries above are mainly presented to emphasize a key difference with the automata that will be introduced in Chapter 4. For now, suffice to say that for classical automata, the corollaries have the same effect as their related definitions; however for assertion automata, where the symbol alphabet is *not* mutually exclusive, the corollaries are not strong enough.

It will be assumed that a procedure for transforming a DFA into a complete DFA can be easily devised. In classical automata, this goal can be accomplished by adding a dead state to a non-complete automaton, then adding transitions to the dead state for all unused symbols in each state's outgoing edges, and then adding loopback edges to the dead state for all symbols. The following example illustrates the nuances between DFAs and complete DFAs.

**Example 2.1:** The regular expression $a(b*)a$ describes the pattern of all strings that start with the character $a$, followed by any number of $b$s, and finishing with an $a$. Figure 2.6 a) shows a DFA corresponding to the given regular expression. This DFA is consistent with Corollary 2.4 for deterministic automata, because no state has two outgoing edges with the same symbol.

Figure 2.6 b) shows a complete DFA that accepts the same language as the DFA in part a). In each state, precisely one outgoing transition always takes place when a character is received.

Converting an NFA to a DFA requires that $\epsilon$ transitions be removed. The procedure for removing $\epsilon$-transitions is based on $\epsilon$-closure [101, 172]. Since $\epsilon$ transitions will not be used in the automata developed in this work, $\epsilon$-removal will be not be treated further.

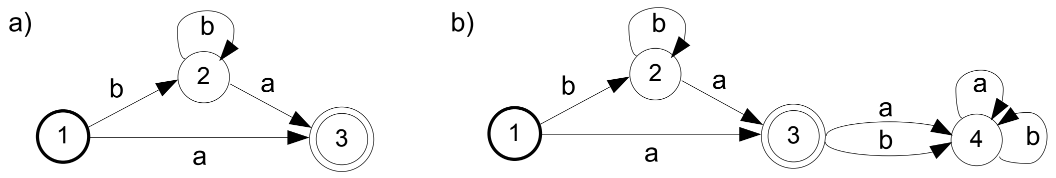


**Figure 2.6:** Determinization does not imply completeness. a) DFA for $a(b*)a$, b) complete DFA for $a(b*)a$.

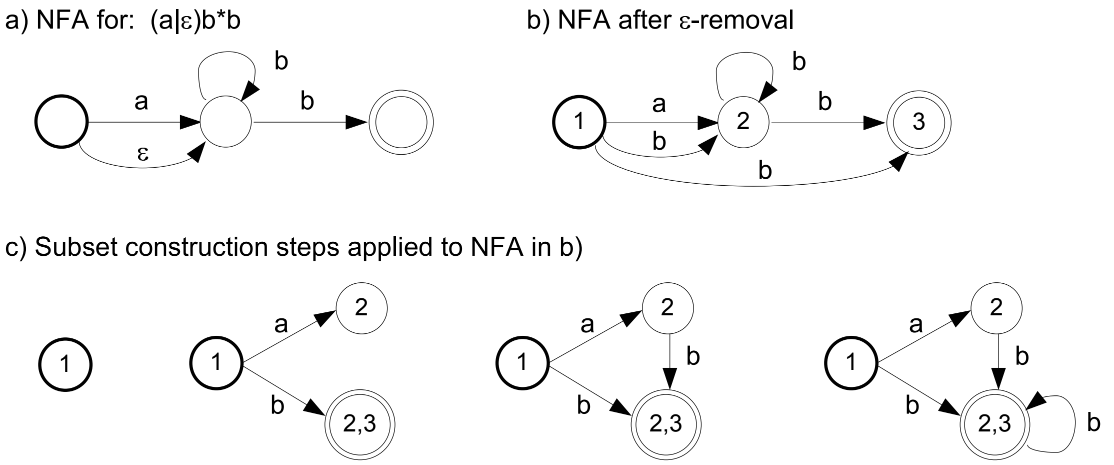


**Figure 2.7:** NFA to DFA example using epsilon removal and subset construction.

An $\epsilon$-free NFA is converted into an equivalent DFA using the *subset construction* technique [101, 172]. Subset construction is the central procedure in the *determinization algorithm*. In a deterministic automaton, at most one outgoing transition must take place in a given state, for a given symbol. In order for multiple outgoing transitions with the same symbol to activate a single destination state, a state in the DFA represents a *subset* of states of the NFA. This way, the criterion for single destination states is respected. Subset construction yields a DFA that has in the worst case an exponential number of states compared to the original NFA. The following example illustrates the conversion of an NFA with $\epsilon$ transitions into an equivalent DFA.

**Example 2.2:** The starting point for the determinization example is the regular expression $(a|\epsilon)b*b$, which describes the pattern consisting of one or more $b$s optionally preceded by an $a$ (the Kleene star applies only to $b$). Figure 2.7 a) shows the NFA corresponding to this regular expression. Figure 2.7 b) shows the effect of $\epsilon$-removal, which in this case can be accomplished by replicating the outgoing edges of the second state in the first state, and removing the $\epsilon$ edge.

Figure 2.7 c) shows the steps taken by the subset construction technique to determinize the automaton. The algorithm first starts by building the initial state of the DFA, which is directly the initial state of the $\epsilon$-free NFA. Since there are two outgoing edges with symbol $b$ in that NFA, leading to states 2 and 3, a new state in the DFA is created for this subset, and is labeled "2,3". The single edge with $b$ in state 1 in the DFA now adheres to the conditions for determinism in Corollary 2.4. State "2,3" is marked as a final state because at least one of the states in the subset is a final state in the $\epsilon$-free NFA. The single outgoing edge with $a$ is not affected and appears similarly to its $\epsilon$-free NFA counterpart. In state 2 in the $\epsilon$-free NFA, $b$ leads to the subset $\{2,3\}$ of states, thus in state 2 in the DFA, the edge with $b$ leads to state "2,3". State "2,3" is handled next: when both states 2 and 3 from the $\epsilon$-free NFA are considered as a whole, edges with $b$ lead to states 2 and 3. A state for this subset already exists, and the self-loop with $b$ is created. The automaton at the bottom right is therefore the equivalent DFA for the NFA at the top left, and illustrates a simple case of $\epsilon$-removal and subset construction.

An operator that does not usually appear in the definition of REs is complementation. If $r$ is a regular expression and $A$ is the equivalent automaton, and $L(A)$ is the language accepted by the automaton (hence $L(r)$ also), then the complemented automaton $\overline{A}$ accepts the language $\overline{L}(A)$, where $\overline{L} = \Sigma * - L$. The full language $\Sigma *$ represents every possible string that can be formed using the alphabet $\Sigma$. The complemented automaton $\overline{A}$ can be computed from $A$ using the following algorithm:

1. Determinize $A$;
2. Make $A$ complete;
3. Complement the final states: $F \leftarrow Q - F$.

The determinization and completion of automata are described here without showing full algorithms. These types of algorithms are formally stated and extensively covered in the automata theory used for PSL checkers in Chapter 4. An example showing the effect of the automaton complementation algorithm above is shown next, as a continuation of Example 2.2.

**Example 2.3:** If the NFA in Figure 2.7 a) is to be complemented, after step one of the complementation algorithm above, the right-most automaton in Figure 2.7 c) is produced, as was explained in the determinization example (Example 2.2).

In step two of the complementation algorithm, the completion adds a dead state (state 3) and the necessary edges so that a complete DFA is produced. This is

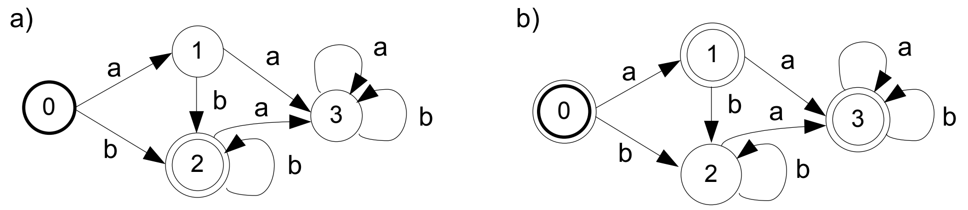


**Figure 2.8:** Complementation example, as a continuation of Example 2.2. a) after completion, b) the resulting complemented automaton.

shown in Figure 2.8 a). The last step of the complementation algorithm involves complementing the set of final states such that final states become non-final states and *vice versa*. The resulting automaton that accepts the language $\overline{L}((a|\epsilon)b{*}b)$ is shown in Figure 2.8 b). This corresponds to all the strings except those consisting of one or more $b$s optionally preceded by an $a$.

Another operator not typically used in regular expressions is intersection. If $r_1$ and $r_2$ are REs, then $r_1\&\&r_2$ is a RE, where $L(r_1\&\&r_2) = L(r_1) \cap L(r_2)$. The $\cap$ symbol represents set intersection. The double ampersand notation was chosen here to more closely relate to the length matching intersection operator found in PSL. The intersection of two regular expressions corresponds to the strings that are in both languages of the REs. It should be noted that the intersection and complementation of regular languages also yield regular languages.

In automaton form, the intersection operator is implemented by building a product automaton from both argument automata. This product automaton is built by simultaneously traversing both argument automata and exploring all paths that have common symbols. In the product construction, a state is labeled using an ordered pair $(i, j)$ where $i$ represents a state from the first argument automaton and $j$ is a state in the second automaton. The algorithm starts by creating an initial state that references both initial states in the input automata. From there, all transitions with a given symbol that simultaneously advance both automata are explored and used to construct new states and edges in the product automaton. A new state is a final state only if both referenced states are final states in their respective automata. In the worst case, the product construction can create an automaton with $mn$ states, where $m$ and $n$ are the number of states in both argument automata.

Some RE specifications, such as POSIX regular expressions, add extra operators to simplify the writing of REs. The "." is used to match any symbol, the "?" is used to match zero or one instance of a symbol, the "+" matches one or more instances and a range of characters can be easily specified in square brackets such as [0-9], which

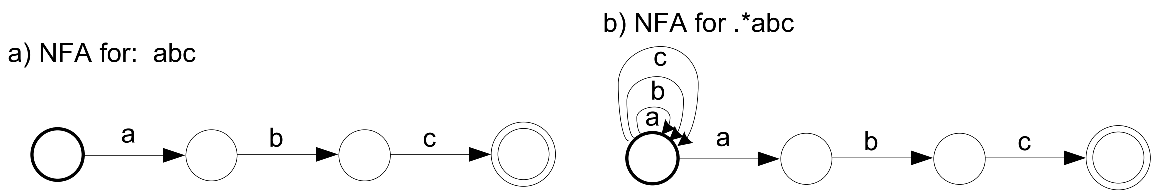


**Figure 2.9:** Continuous matching example. a) single match starting in the first input letter, b) continuous matching.

matches any numeric character.

When using automata to perform pattern matching, if the regular expression describes a pattern to be matched anywhere in the input stream, a prefix repetition becomes necessary. For example, Figure 2.9 a) shows the automaton for the regular expression *abc*. The automaton's initial state is deactivated after the first input letter is processed because there are no loopback edges to re-activate it. The pattern *abc* will only be checked starting in the first input letter. In order for *every* occurrence of the pattern to be matched, the regular expression $.*abc$ should be used, where the "." matches any symbol in the alphabet $\Sigma$. In Figure 2.9 b) the added prefix "$.*$" in effect causes the automaton to be retriggered at every step, and allows the intended expression *abc* to be matched starting at any point in the input stream.

The next section presents the main features of the PSL language. As will be shown, regular expressions play a key role in creating temporal regular expressions over Boolean propositions.

## 2.4 The Property Specification Language

The Property Specification Language is formally defined as the IEEE 1850 standard [111]. More pragmatic treatments of PSL are presented in related textbooks [53, 67, 74, 148]. The presentation of PSL contained in this section is based on Appendix A in the PSL specification [111], which describes the formal syntax of the PSL language in BNF (Backus-Naur Form). The semantics of each operator is explained informally in this section, and is formally specified in Appendix B in the PSL specification [111]. The following considerations were made for the PSL used and presented in this work:

- The Optional Branching Extensions (OBE) are not suitable for dynamic verification and are omitted from consideration (section 4.4.3 in the PSL specification [111]).
- The simple subset restrictions for "simulatable" PSL were taken into account

**Table 2.1:** Commonly used Verilog language operators.

| | |
|---|---|
| Logical negation | ! |
| Bitwise negation | ∼ |
| Logical equality | == |
| Logical inequality | != |
| Bitwise conjunction (and) | & |
| Bitwise disjunction (or) | \| |
| Logical conjunction (and) | && |
| Logical disjunction (or) | \|\| |

and applied directly to the language definitions herein (from section 4.4.4 in the PSL specification [111]). These restrictions are necessary for PSL to be used in dynamic verification.

- Expressions appearing in bounds for ranges and numbers are restricted to integers in this work, as opposed to statically computable expressions. Adding support for expressions involves additional features in the parser front-end, and does not affect the algorithms used in the checker generator.
- Sequence and property instantiations are omitted for simplicity. In the current version of the checker generator, non-parameterized sequence and property declarations and instantiations are supported.
- The two directives that apply to dynamic verification are implemented, namely assert and cover. Other directives are intended for formal methods and are not implemented, with the exception of assume, which is implemented in the same way as assert [74].
- The clocking operator in sequences and properties is omitted since its implementation in the checker generator is beyond the scope of this work.
- Although the PSL language defines flavor macros for supporting many HDLs, the Verilog flavor will be used throughout this work, and all necessary operators will be shown in the Verilog language.

The Verilog HDL language is used to specify the Boolean expressions that form the core of PSL's Verilog flavor. A set of commonly used Verilog operators is shown in Table 2.1, in decreasing order of operator precedence; the full BNF syntax is available in standard Verilog textbooks [22]. In the pseudo-code algorithms in upcoming chapters, comments are specified in the same way as C and Verilog comments, by using the double forward slash //.

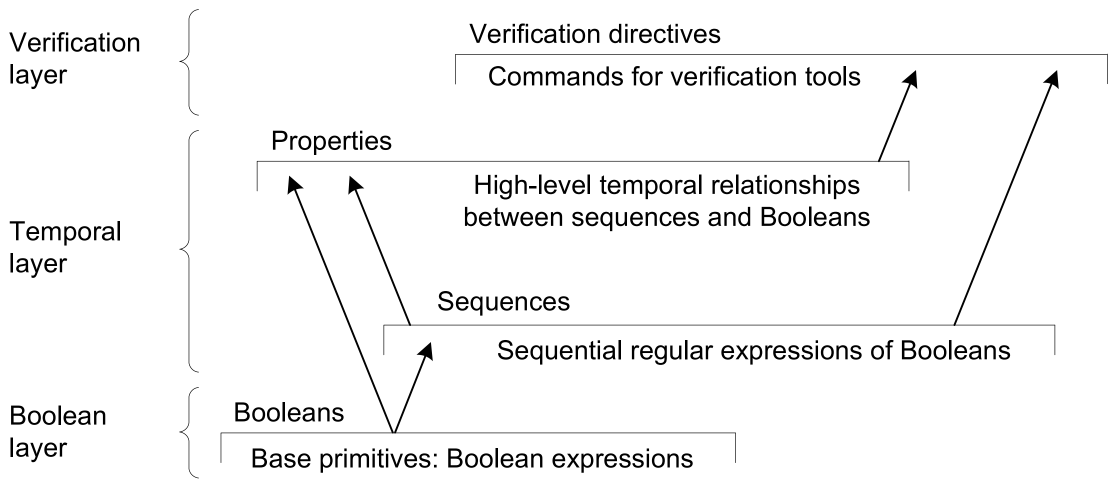

Verification layer
Temporal layer
Boolean layer
Verification directives
Commands for verification tools
Properties
High-level temporal relationships between sequences and Booleans
Sequences
Sequential regular expressions of Booleans
Booleans
Base primitives: Boolean expressions

**Figure 2.10:** PSL language hierarchy.

Constants in Verilog can be specified directly as numbers, with optional prefixes for indicating the base and the size of the constant. For example, the constant 4'b1010 is a four-bit binary number, whereas 'h55 is an unsized hexadecimal number. Verilog parameters are handled by the checker generator and can even be used to automatically create parameterized assertion checkers.

PSL is intended for both formal and dynamic verification; however, since it is mostly used for dynamic verification with the checker generator, some operators are best explained in the context of run-time verification. The top level PSL directives that are used to capture design intent are presented at the end of this section, and the elements used to define them are defined gradually starting with Booleans, and then sequences and properties.

The Property Specification Language is defined in four layers, namely the Boolean layer, the temporal layer, the verification layer and the modeling layer. The modeling layer allows the specification of additional signals and variables, as well as the modeling of design inputs for use in formal verification. The modeling layer is not treated in this work since it has no substantial effect on the assertions themselves.

The *Boolean layer* in PSL is built around the Boolean expressions of the underlying HDL, which reference a set of Boolean-valued signals (Boolean propositions). Italicized prefixes indicate an additional constraint on a type.

**Definition 2.7:** If "Number" is a nonnegative integer then *Booleans* are defined as follows in PSL ("Sequence" is defined further in this section):
Boolean ::=
    *boolean*_Expression

Expression ::=
    *Verilog*_Expression
  | Boolean -> Boolean
  | Boolean <-> Boolean
  | true
  | false
  | Built-in_function
Built-in_function ::=
    prev(Expression)
  | prev(Expression, Number)
  | rose(*bit*_Expression)
  | fell(*bit*_Expression)
  | onehot(*bit_ vector*_Expression)
  | onehot0(*bit_ vector*_Expression)
  | stable(Expression)
  | ended(Sequence)

Although not formally part of the specification, symbols true and false are used to simplify the notation in this work, and are defined respectively as 1'b1 and 1'b0, in Verilog notation. Built-in functions allow the specification of single or multiple cycle conditions but are not part of PSL's temporal layer.

The prev() operator returns the previous value of the expression argument, one cycle or a specified number of clock cycles previous to the current cycle. The rose(), fell() and stable() operators compare the values of their arguments in the current cycle with the previous cycle. The onehot() evaluates to true if *exactly* one bit in the argument is at logic-1; the onehot0() operator evaluates to true if *at most* one bit is at logic-1. The ended() operator evaluates to true every time its sequence argument is matched.

Other built-in operators also exist, such as next(), isunknown(), countones(), nondet(), nondet_vector(), but are omitted from Definition 2.7 for a variety of reasons. The next() operator is non causal and is not suitable for dynamic verification. The isunknown() function is used to report any values that are not at logic-0 or logic-1, and is not implemented in the checker generator. The countones() function could be implemented using a population count algorithm; however, instantiating such complex HDL code could drastically affect the performance of the checker generator, thus the function is purposely not implemented. The nondet() and nondet_vector() operators represent a nondeterministic choice of a value within a value set, and are also omitted for hardware implementation reasons.

Sequential-Extended Regular Expressions (SEREs) and Sequences are used to specify temporal chains of events of Boolean expressions, and are at the core of PSL's *temporal layer*. Sequences are built upon Booleans, and are a temporal version of regular expressions. In the remainder of the text, the term *sequence* (not initially capitalized) will be used to refer to all of the items in the definition below, in a general manner; the term *Sequence* (initially capitalized) will refer to the actual syntax item in the definition.

**Definition 2.8:** If "Number" represents a nonnegative integer, and normal typeface brackets [] represent an optional parameter, then *Sequences* and *SEREs* are defined as follows in PSL:

```
Sequence ::=
    repeated_SERE
  | {SERE}
SERE ::=
    Boolean
  | Sequence
  | SERE ; SERE
  | SERE : SERE
  | compound_SERE
compound_SERE ::=
    Sequence
  | compound_SERE | compound_SERE
  | compound_SERE & compound_SERE
  | compound_SERE && compound_SERE
  | compound_SERE within compound_SERE
repeated_SERE ::=
    Boolean[*[Count]]
  | Sequence[*[Count]]
  | [*[Count]]
  | Boolean[+]
  | Sequence[+]
  | [+]
  | Boolean[= Count]
  | Boolean[->[positive_Count]]
Count ::=
```

Number
| Number : Number
| Number : inf

The operators for sequences and SEREs in the definition above are not listed in order of operator precedence. Section 4.2.3.2 in the PSL specification [111] lists the relative precedence, from highest to lowest: [* ] and [+] and [–> ] and [= ], within, & and &&, |, :, ;.

Some aspects of SERE notation are equivalent to conventional regular expressions: the [*] operator is a repetition of zero or more instances (Kleene closure), the | operator corresponds to SERE disjunction (choice), and ; represents temporal concatenation. The curly brackets are equivalent to parentheses in regular expressions. In the context of SEREs for property verification, concatenation of two Boolean expressions $b_l; b_r$ produces a match when the Boolean expression $b_l$ evaluates to true in one cycle and $b_r$ evaluates to true in the next cycle.

Other SERE operators are seldom used, or have no equivalent in conventional REs. The : operator denotes *SERE fusion*, which is a concatenation in which the last Boolean expression occurring in the first SERE must intersect (i.e. both are true) with the first Boolean primitive occurring in the second SERE. Empty SEREs in either side do not result in a match. The *length matching SERE intersection* operator && requires that both argument SEREs occur, and that both SEREs start and terminate at the same time. The single & represents non-length matching intersection, whereby SEREs must start at the same time but do not necessarily end at the same time. The matching thus occurs when the longer of the two SEREs is matched, provided the shorter SERE was matched. The within operator causes a match when a sequence occurs within another sequence. The shorter sequence starts after (or at the same time) and terminates before (or at the same time), compared to the longer one.

In repeated SEREs, the [*Count] operator can be used to model a fixed-length repetition or a repetition range. A successful range match occurs when the expression being repeated is matched a number of times contained in the specified interval. The [+] symbol indicates a repetition of one or more instances. When the various forms of [*] repetition or the [+] repetition are used without a Boolean or Sequence, the Boolean true is implicitly assumed.

The [–>] operator is known as goto repetition, and causes a matching of its Boolean argument at its first occurrence. A fixed-length goto repetition or a range of goto repetitions can also be specified, whereby the integers used must be greater than zero. The [=] corresponds to non-consecutive repetition, which is similar to a goto

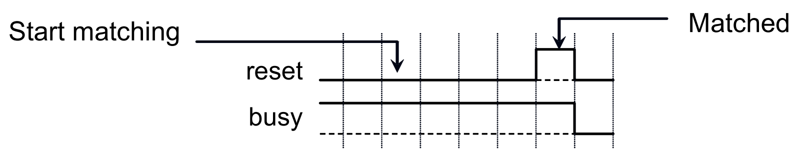


**Figure 2.11:** Sequence matching for $\{\{busy[*]\}\ \&\&\ \{reset[->]\}\}$.

repetition that can cause additional matches for extended cycles where the Boolean remains false.

**Example 2.4:** The following sequence causes a match at the first occurrence (logic-1) of the reset signal, provided that the busy signal is continually asserted.

$$\{\{busy[*]\}\ \&\&\ \{reset[->]\}\}$$

Figure 2.11 illustrates how the sequence above causes a match for an arbitrary start condition. This start condition could originate from the left side argument of a concatenation, if the above sequence was used as the right side in the concatenation. Even though the start condition occurs before the reset becomes true, the goto repetition extends the matching as needed. The length matching intersection with the variable-length left side (with busy) was also observed and helped to produce the successful match.

In reality, only a subset of the operators in Definition 2.8 are required to completely specify SEREs and sequences. The following equations are derived from the sugaring rules in Appendix B in the PSL specification [111], and help to understand the semantics of the more convoluted operators mentioned in the previous paragraphs ($b$ is a Boolean, $i, j$ are positive integers and $k, l$ are nonnegative integers)[1]:

$$\begin{aligned}
b[*k] &\overset{\text{def}}{=} b;b;\ \ldots;b \quad (k \text{ times})\\
b[*i{:}j] &\overset{\text{def}}{=} b[*i]\ |\ \ldots\ |\ b[*j]\\
b[->] &\overset{\text{def}}{=} {\sim}b[*];b\\
b[->k] &\overset{\text{def}}{=} \{{\sim}b[*];b\}[*k]\\
b[->k{:}l] &\overset{\text{def}}{=} b[->k]\ |\ \ldots\ |\ b[->l]\\
b[=i] &\overset{\text{def}}{=} \{{\sim}b[*];b\}[*i];\ {\sim}b[*]\\
b[=i{:}j] &\overset{\text{def}}{=} b[=i]\ |\ \ldots\ |\ b[=j]
\end{aligned}$$

[1]The intervals $[i] \ldots [j]$ could also be written as $[i]$, $[i{+}1] \ldots [j]$.

The [*0] operator is known as the *empty SERE* and is equivalent to the $\epsilon$ expression from conventional REs. The empty SERE is a primitive that spans no clock cycles. When used as a sequence, the empty SERE is also referred to as the empty sequence. Definition 2.8 also allows the specification of a *null sequence*, but not directly as an operator. For example, the following sequence reduces to the null sequence $\emptyset$ because the length matching intersection of two SEREs of different lengths can not produce a match:

{ {true} && {true[*2]} }

In the PSL specification [111], no symbol is introduced for the null sequence (nor for the null SERE) as it can not be specified directly by the user. The null sequence is similar to the $\emptyset$ in regular expressions and their languages (Definition 2.1), and the same symbol will be used in the next chapters to describe such a sequence. The null sequence does not match anything, while the empty sequence represents an instantaneous match (an empty match is a match nonetheless).

**Definition 2.9:** The empty sequence and the null sequence that can be specified using the syntax in Definition 2.8, either directly or indirectly, as described in the previous paragraphs, are known as *degenerate sequences* [110].

PSL's *temporal layer* also defines properties built on sequences and Booleans. The PSL foundation language properties are shown below in the Verilog flavor, and are presented with the simple subset modifications for dynamic verification [111]. Similarly to SEREs, properties are built from a reasonably compact set of operators to which sugaring operators are also added. However, because the simple subset imposes many modifications to the arguments of properties, the distinction between sugaring and base operators becomes much less relevant and will not be made.

**Definition 2.10:** If "Number" represents a nonnegative integer, then PSL *properties* and *foundation language properties* are defined as follows in the simple subset:

Property ::=
  forall *ident* in boolean: Property
  | forall *ident* in {Range}: Property
  | FL_Property
FL_Property ::=
  Boolean
  | (FL_Property)
  | Sequence !

| Sequence
| FL_Property abort Boolean
| ! Boolean
| FL_Property && FL_Property
| Boolean || FL_Property
| Boolean -> FL_Property
| Boolean <-> Boolean
| always FL_Property
| never Sequence
| next FL_Property
| next! FL_Property
| eventually! Sequence
| FL_Property until! Boolean
| FL_Property until Boolean
| FL_Property until!_ Boolean
| FL_Property until_ Boolean
| Boolean before! Boolean
| Boolean before Boolean
| Boolean before!_ Boolean
| Boolean before_ Boolean
| next[Number](FL_Property)
| next![Number](FL_Property)
| next_a[Range](FL_Property)
| next_a![Range](FL_Property)
| next_e[Range](Boolean)
| next_e![Range](Boolean)
| next_event!(Boolean)(FL_Property)
| next_event(Boolean)(FL_Property)
| next_event!(Boolean)[*positive*_Number](FL_Property)
| next_event(Boolean)[*positive*_Number](FL_Property)
| next_event_a!(Boolean)[*positive*_Range](FL_Property)
| next_event_a(Boolean)[*positive*_Range](FL_Property)
| next_event_e!(Boolean)[*positive*_Range](Boolean)
| next_event_e(Boolean)[*positive*_Range](Boolean)
| Sequence |-> FL_Property
| Sequence |=> FL_Property

Range ::=
    Number : Number

The term *property* (not capitalized) will be used to refer to all of the items in the definition, in a general manner. The term *Property* (capitalized) refers to the actual syntax item in the above definition. Property instantiation and sequence instantiation can be implemented in the parser front-end, and are omitted from the presentation because they do not add any computational complexity to checker generation.

The properties in Definition 2.10 are not listed in order of operator precedence. Section 4.2.3.2 in the PSL specification [111] lists the relative precedence, from highest to lowest: abort, the next family and eventually!, the until and before families, |-> and |=>, -> and <->, always and never.

The forall operator has more variations than what was shown above; however, the two versions in the definition are used to convey the main idea behind replicated properties. The *ident* parameter is simply a unique identifier that is used as a variable in the argument property. The forall operator replicates the property with each successive version taking a different value for the *ident* variable. Each instantiation of the replicated property is expected to hold, thus the replication creates a multiple conjunction of properties.

Since a braced SERE is a valid sequence, the foundation language property:

{SERE}(FL_Property)

was not listed, as it is semantically equivalent to the foundation language property shown below:

{SERE} |-> FL_Property

The standard LTL operators X, X!, G, F, U and W are equivalent to operators next, next!, always, eventually!, until! and until, respectively, and for simplicity were omitted in Definition 2.10. Furthermore, as indicated in the working group issues for PSL [112], the left side arguments of the operators until_ and until!_ do not need to be restricted to Boolean expressions in the simple subset [111]. This change is expected to appear in the next revision of the PSL standard. The async_abort and the sync_abort are treated the same way as abort in the checker generator, and were omitted. The never and eventually! operators also accept a direct Boolean as their arguments, but for simplicity was omitted from the syntax above.

Parentheses around a property are used only for grouping. The definition also shows that Booleans and sequences can be used directly as properties, thereby indicating that the sequence or Boolean expression is expected to be matched, and that a non-occurrence constitutes a failure of the property. The matching is weak, meaning that if the end of execution occurs before the matching is complete, then the property holds. A sequence can be made to be a strong sequence using the ! operator, thereby specifying not only that the sequence should be matched, but that it should be matched before the end of execution.

The abort operator can be used to release an obligation on a property, when a given Boolean condition occurs. This operator is particularly useful for re-initializing properties when an external reset occurs in the design under verification. This prevents the checker from continuing to monitor a property across a reset event.

In the simple subset, negation and equivalency of properties are not defined and must be performed only with Booleans. These operators have the same semantics as in the Boolean layer, and the resulting top-level Boolean is expected to hold as a property. Therefore, if the Boolean is not matched then the property fails.

Property implication and property disjunction allow at most one of their arguments to be a property. In the case of the implication operator, the antecedent must be a Boolean. If the antecedent of the implication occurs, then the consequent property is expected to hold. If the antecedent does not occur, the consequent property holds vacuously. A behavior similar to implication also exists for disjunction. If the Boolean is false then the argument property is expected to hold, and if the Boolean is true then the property holds. In the definition of properties, even though the Boolean is shown as the left side argument of ||, permuted arguments are also acceptable. Property conjunction &&, not to be confused with SERE intersection, is used to specify that two properties must both hold.

The always operator specifies how its argument property should behave. When the always property receives an activation it will continually activate (retrigger) its argument property, consequently causing it to always be checked. The activation comes from the next outermost expression in the given PSL statement. The retriggering aspect of this temporal operator is analogous to the sliding window [57] used in certain string matching algorithms, where the matching is to be performed at every position afterwards.

**Example 2.5:** This example shows different types of activations and their effect on the always operator.

$$\text{assert } f\_p \tag{2.4}$$

$$\text{assert always } f\_p \tag{2.5}$$

$$\text{assert } b \text{ -> (always } f\_p) \tag{2.6}$$

In (2.4), the property $f\_p$ is only activated in the initial clock cycle (the first one after the reset is released). The property is only expected to hold starting in that cycle, and depending on its temporal length and the circuit conditions, it may signal an assertion failure at that time, or any amount of clock cycles in the future. In (2.5), the property always $f\_p$ is itself activated on the initial clock cycle, and proceeds to continually activate the checking of property $f\_p$. The property is thus expected to hold starting in all clock cycles. In (2.6), the temporal implication causes the always sub-property to be activated only when the antecedent $b$ is observed. Hence, once $b$ is observed the property $f\_p$ is expected to hold in this clock cycle and all cycles to come.

The never operator behaves similarly, with the exception that it continually extends the matching of its sequence argument, relative to its activation. Once activated, any future matching of the sequence causes the property to fail. The next operator starts a checking of its property argument in the cycle following its own activation. This is a weak property, meaning that if the next cycle does not occur then the property holds. The strong version of this operator, namely next!, does not allow the end of execution to occur in the next cycle. In other words, the next cycle must be a valid execution cycle and the argument property must hold, in order for the next property to hold. Incidentally, next[0]($f\_p$) is equivalent to $f\_p$.

The eventually! property states that its argument sequence will be observed before the end of execution occurs. Once again, activations affect the behavior of this property, as shown below.

$$\text{assert eventually! } \{b;c;d\};$$
$$\text{assert } a \text{ -> eventually! } \{b;c;d\};$$

In the first assertion above, the sequence is expected to be observed before the end-of-execution. In the second example, once $a$ is observed, the sequence is expected to be observed before the end-of-execution.

The end of execution is a special signal that must be provided by the user to indicate that no further clock cycles will occur. This signal is not part of the PSL specification, and is used in dynamic verification tools to implement strong properties. All properties with the ! symbol are temporally strong properties. Weak properties do not require the end of execution signal.

The until family of properties in Definition 2.10 cause the continual checking of their argument property until the releasing Boolean occurs. In the overlapped versions (with the _), the argument property is also checked in the clock cycle were the Boolean occurs. In the strong versions (with the !), the Boolean *must* occur before the end of execution.

The before family of operators specify how two Booleans should relate temporally to one another. In the first of such operators in Definition 2.10, the left side Boolean should occur strictly before the right side Boolean, or else the property fails. In the overlapped versions, the left side Boolean is also allowed to occur in the same clock cycle as the right side Boolean. In the strong versions, the right side Boolean *must* occur before the end of execution.

The next[]() properties are extensions of the next properties mentioned previously, with a parameter for specifying the $n^{\text{th}}$ next cycle. This applies to both the weak and the strong versions. The next_a properties cause the checking of the argument property in a range of next cycles, specified with a lower and upper bound integer. The next_e properties apply only to Booleans, and are used to indicate that the given Boolean must be observed at least once within a specified range of next clock cycles.

Up to this point, a total of eight variations of next properties have been encountered. The basic "unit of measurement" that is implied when referring to "next" is the clock cycle. The remaining eight next-type properties in Definition 2.10 are based on a different unit, namely the next *event*. The next_event properties are similar to the next properties, except that the "next" being referred to is not clock cycles but rather the occurrence of a given Boolean. For example, next_event_a! is used to specify that an argument property must be true within a range of next occurrences of an argument Boolean, and that all next occurrences of the Booleans specified in the range must occur before the end of execution. A subtlety worth mentioning is that the next event of a Boolean can be in the current cycle.

**Example 2.6:** This example illustrates the behavior of two forms of next properties.

$$\mathsf{next}[2](\sim busy)$$

$$\mathsf{next\_event\_a}(flag)[2{:}4](\sim busy)$$

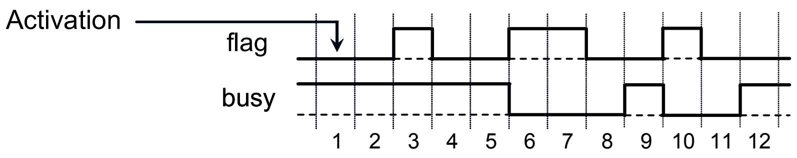


**Figure 2.12:** Traces for next properties in Example 2.6.

The first property makes use of the plain next operator, with a parameter to indicate that the argument property, in this case ∼*busy*, must hold in the second next cycle, with respect to when the property is activated. In the example trace in Figure 2.12, the property does not hold because its argument property does not hold in cycle 3 (*busy* is not false). The second property indicates that its argument property (∼*busy*) must hold in all of the next two to four occurrences of the *flag* event. In the example trace in Figure 2.12, this property is true because in cycles 6, 7 and 10, *busy* is false. In these examples the activation would come from the parent operator when the properties are used in more complex expressions. For example, if the two properties above are used in place of $p$ in:

always $b$ −> $p$

then the activation being referred to in the figure corresponds to an occurrence of the Boolean $b$.

The two forms of temporal implications (|−> and |=>) are referred to as overlapped and non-overlapped suffix implication respectively. In overlapped suffix implication, for every matching of the antecedent sequence, the consequent property is expected to hold. The property must hold starting in the last cycle of all the antecedent sequence's matches. In the non-overlapped suffix implication, for each successful matching of the antecedent sequence the consequent property is expected to hold starting in the cycle after the last cycle of the antecedent's match.

Properties and sequences as such accomplish nothing for verification and have no effect. The *verification layer* in PSL is used to instruct tools on what to do with these properties and sequences. The two most often used verification directives are shown below.

**Definition 2.11:** PSL *verification directives* are defined as follows:
Verification_Directive ::=
  assert Property ;
  | cover Sequence ;

The assert directive instructs a verification tool to verify that the argument property holds, whether it is checked formally or in run-time verification. The cover directive instructs a tool to verify that the argument sequence must occur at least once during the verification process. Although the word *assertion* should strictly be used to refer to an assert directive, the term assertion is also used more generally to encompass all verification directives.

To specify the clock reference, the default clock must be declared before any verification directive can be used. In this work, the scope of the clock declaration extends to all directives below it, until it is redefined; however, according to the PSL specification, at most one clock declaration can be specified in a vunit. The *clock_signal* used usually corresponds to one of the clock domains in the source design.

**Definition 2.12:** In PSL, the *default clock declaration* can be specified using either of the following statements:
Clock_Declaration ::=
    default clock = (posedge *clock_signal*);
  | default clock = (negedge *clock_signal*);

The *items* in both preceding definitions are grouped in a vunit when PSL commands are to be specified in a separate file. The vunit's prototype is used to bind the items to a source design. PSL items can also be declared using comments directly in the source design's code (in-line assertions). In both cases, all signal names used in the items must be valid signal names from the source design.

PSL does not have a formally defined run-time semantics for its interpretation. The language reference manual [111] specifies only the conditions under which properties pass or fail, and the run-time interpretation of PSL as used in a checker generator or a simulator is not imposed. For example, for a given assertion, the semantics of PSL described in the reference manual dictate the conditions under which the assertion holds, but does not mention how/when/where to report failures, and how many failures must be reported.

The run-time semantics used in the checker generator are sometimes different than those used in simulators capable of interpreting PSL. Interpreting assertions by specialized hardware is different than interpreting assertions in a simulation kernel, where processes and dynamic memory allocation are readily available. In both cases, efficiency is a driving parameter in design choices for PSL interpreters. The run-time semantics resulting from the implementation of PSL in Chapter 5 is well suited to hardware checkers. Dynamic property checking semantics can also be derived from the operational semantics presented by Claessen and Martensson [50].

Some implementations of sequences and properties will be described using rewrite rules in Chapter 5. Rewrite rule have the form $x \rightarrow y$, where the operator in the left side ($x$) is syntactically rewritten to the expression in the right side ($y$). This is normally done in a tool's front-end to rewrite an operator to another one that is already supported, such that the left side operator does not have to be explicitly implemented in the tool's kernel. Proper care must be taken when devising a set of rewrite rules, such that the set is terminating [62].

**Definition 2.13:** A set of rewrite rules is said to be *terminating* if there is no infinite sequence of application of the rules.

To complete the presentation of the PSL language, a summary example illustrates how assertions can be used in practice.

**Example 2.7:** To illustrate the use of assertions in a more complete example, a width-parameterized up-down counter is verified. The counter has a signal to enable counting (*en_ud*), and another signal to control the counter direction (*up_ndown*). The counter can also be loaded with an external value (*load*), which has higher priority than the counting-enable signal. The load is performed when *en_load* is true, whereby the counter is synchronously loaded with the *load* value in effect at that time. The loaded value then appears on the counter's output (*cnt*) in the next cycle. The Verilog parameter *width* is used to control the size of the signal vectors *cnt* and *load*. The counter is designed in RTL using the *clk* signal as a reference for its sequential elements (flip-flops). The four assertions shown below are used to perform the verification, although more assertions could be added to cover different features of the counter.

**UDCNT_ASRa - UDCNT_ASRd**:

```
assert always {~en_load & ~en_ud} |=> stable(cnt);
assert always en_load -> next (cnt == prev(load));
assert always ~en_load ->
                    next (!(cnt==~prev(cnt) && cnt[width-1]==cnt[0]));
assert never (~en_load & ~en_ud)[*10];
```

The first two assertions are used to ensure the proper operation of the counter. If the counter is not being loaded with a new value and it is also not counting, then the output value should not change. In the second assertion the loading process is verified to ensure that the new value takes effect. The third and fourth assertions are not based on properties of the counter as such, instead they describe the way the counter is actually used. In this example, the use of the counter is such that rollover should never occur (third assertion), and the environment should be updating the counter periodically (fourth assertion). Assertions can be used to verify the intrinsic operation of a circuit, and also to verify that the circuit is properly used in its intended environment.

When considering only the assertions above, the signal dimensions can not be deduced. For example, *cnt* and *load* are actually declared with dimensions [*width*-1:0] in the source design. In order for the checker generator to produce the correct circuit for prev(*cnt*) for example, the vector width of the *cnt* signal must be known. It is for these reasons that the checker generator must also have access to the source design when generating assertion checkers, as was illustrated in Figure 2.2. The code for the PSL assertions, the source design and the generated checkers in this example is shown in Appendix A.

In Example 2.7, a checker generator would be used to create a monitoring circuit from the assertions for inclusion into hardware emulation, silicon debug or post-fabrication diagnosis. Special considerations such as assertion multiplexing and assertion grouping must be made when a *large* amount of checkers are to be emulated and probed. Although in this example assertions may not be required by experienced designers to help verify such a simple counter, the benefits of assertions are quickly amplified in larger designs.

# Chapter 3

# Related Research

This section presents research related to the automated generation of checkers from assertions. The first two sections deal with two fundamentally different approaches to checker generation, namely the modular and automata-based approaches. Although automata-based methods are currently used in the checker generator, the modular approach was used in the initial versions of the tool [29]. Formal verification and model checking techniques make heavy use of various types of automata, and these are surveyed in Section 3.3.

Section 3.4 describes how assertions are typically used in simulators and emulators. Assertion checkers also play a key role in silicon debugging scenarios, and different techniques used to improve the usability of assertions are explored in Section 3.5. Miscellaneous topics are relegated to the final section, where a variety of different languages are also briefly presented.

## 3.1 Modular Approach to Checker Generation

In the modular approach to checker generation, each PSL operator is implemented as its own separate sub-module (or sub-circuit), and is connected to other modules to form a checker for a complete PSL assertion. Typically, the modules have a predefined signaling protocol whereby activations and result signals are exchanged. This often consists of a single wire, or a pair of wires. The main feature of this approach is that each sub-module retains its internal integrity and is used as a black box for the operators it implements. The actual content of the black box is transparent to the other modules to which it is connected. The sub-modules are interconnected according to the syntax tree of the assertion, in a recursive process. Once the complete circuit is constructed, a checker is produced and the output signals can be observed during verification for finding errors.

The HORUS checker generator is being developed at the TIMA-VDS laboratory in France by researchers Borrione and Morin-Allory, and their associates. Besides generating Verilog or VHDL observation monitors, the tool also generates code for test sequence generation [140]. Although their tool is still under development, a number of publications detail their approach to checker generation. These are briefly overviewed in the next two paragraphs.

Implementing SEREs using the modular approach first consists of building a library of sub-modules to represent the different SERE operators, which are then used to build a network according to the syntax tree of the SERE being implemented [134]. The interconnection protocol is based on tokens that are passed from one module to the next. Colored tokens are used when sequences are used as properties (in the consequent of a suffix implication for example). The colored tokens are used to maintain the correspondence between different overlapping activations. Although it is mentioned that all SERE operators are supported, no benchmarks are performed. Furthermore, when multiple concurrent matches are taking place, a large number of colors must be supported in the tokens, which represents a non-negligible hardware overhead in the circuit representation. The token approach shares a commonality with the *assertion threading* debug enhancement developed in Section 6.4.5 because both techniques can help correlate a failure with its given start condition. Although the techniques involved are very different, the parallel between the two is very interesting and was observed by Borrione [23].

Another set of publications from the TIMA group presents a modular implementation of PSL foundation language properties [24, 25, 133, 135]. A pair of interconnect signals is used, and assertions produce a pair of signals that indicate the status of the assertion. The approach consists of developing sub-modules for each property operator and interconnecting the appropriate modules to form a complete checker for an assertion. Moreover, the functional correctness of the methods is proved using formal methods in the PVS proof system. The library of components for property operators is proven correct, and then the interconnection scheme is proven correct, and by induction the checkers that are generated are proven correct. The earlier publications reference Gascard's work [79] for the implementations of SERE, which is reviewed in the next section. Experimental results are reported [24, 25] for properties, whereby the checkers produced are compared to those produced by FoCs v2.02. Although both strong and weak properties are supported, strong properties are not benchmarked. The checkers are implemented using the Synopsys Design Compiler and the size metric reported is based on two-input NAND gates.

Generating interface monitors for verifying proper protocol between design modules is the topic of the research work by Oliveira and Hu [142, 143]. This work is based on conventional regular expressions with an added pipelining operator and storage variables, and does not use PSL nor SVA. The modular scheme is also employed, and activations are passed from one sub-circuit to the next to perform the required pattern matching. Restrictions on pipelining and threading in the monitoring circuits are imposed such that efficient monitors can be built. Experimental results are performed [142] for the AHB and OCP bus protocols, where items from the respective specifications are modeled in the custom language. Only the number of flip-flops is reported in the hardware metrics for the monitor circuits that are produced.

The checker generation of SVA (SystemVerilog Assertions) is explored by Das *et. al.* [60]. Sub-modules corresponding to the different sequence[1] constituents are interconnected with wires labeled “start” and “match”. Sequence operators are classified into subsets that have different synthesis approaches, and a special subset for sequences containing unbounded repetition operators is declared. An indication that the modular approach has its challenges is exemplified by the fact that the authors make mention of cases where expressions from the unbounded subset can not be synthesized into a finite amount of hardware resources. As will be shown in the following chapters, this limitation is not present when automata are used. Synopsys Design Compiler is also used to synthesize a set of benchmark assertions, and the Synopsys VCS simulator is used to compare the checkers with OVA checkers (OpenVera Assertions [166]). In those results, assertions for the ARM AMBA AHB bus interface are used. Among the interesting findings [60] is the description of the “not” operator for matching sequences in the consequent of suffix implications. Separate rules are given to implement the “not” operator for each sequence operator, however no explanations nor insights into the correctness of the proposed rules are given. The run-time semantics produced by the rules is also not addressed. The only SVA properties that are supported are the two forms of suffix implication.

The implementation of SVA checkers is also explored by Pellauer *et. al.* [147]. The “first-match” operator is used as a basis for implementing sequences in the consequent of suffix implications. Checkers are produced in the BlueSpec SystemVerilog language, which is an unclocked language where models are subsequently translated into sequential hardware. The implementation is not fully modular as such, but does share a similar barrier that was observed in other modular approaches. The SERE

[1]SVA also defines sequences, which are similar to PSL sequences.

matching is performed using FSMs (Finite State Machines), whereby a single FSM is used to implement the antecedent in suffix implications, and multiple FSMs are used in the consequent. Since a finite number of FSMs in the consequent are used to process the matches triggered by the antecedent, unbounded repetition is disallowed in the antecedent FSM. The authors also introduce the concept of statistic-gathering assertions by integrating counters within the checkers. A case study on a cache controller is presented, and hardware synthesis results showing the overhead added to the design are measured, both for the normal functional assertions and the statistic-gathering assertions.

Assertion checkers must be able to handle multiple concurrent sequences of events that can overlap temporally. In general, the interconnected module approach shows difficulties for implementing certain operators, especially when dealing with the failure matching of SEREs that use unbounded repetitions and intersection (the [*] and && operators). This difficulty was observed in the first version of the checker generator in this work [29]. Using automata-based methods is one possible way around this problem, and the checker generator developed in this work does not have the limitations mentioned throughout this section.

## 3.2 Automaton Approach to Checker Generation

Regular Expressions (REs) are used in a temporal manner in assertion languages. As a first step to producing hardware assertion checkers, hardware for the matching of regular expressions is first explored. Although hardware RE matchers are not assertion checkers as such, they can still be considered as checkers in some respect.

Among the earliest work in designing custom circuitry for performing hardware RE matching is the compilation REs into integrated circuits by Floyd and Ullman [73]. One of the target applications is the PLA (Programmable Logic Array), which implements logic by interconnecting fixed rows and columns of wires and gates. In another implementation, the circuit's layout structure is directly guided by the hierarchical NFA construction from REs (as shown in Figure 2.5 in the previous chapter). The hierarchical nature of the circuit construction uses signals to interconnect sub-automata, and has similarities to the modular approach; however, the *modular* implementation of RE intersection (not treated) would likely not be possible given that no simple pictorial representation of NFA intersection exists, as does exist for the conventional RE operators (Figure 2.5). The experimental results show that the area of the generated circuits grows linearly with the size of the regular expression.

A more modern hardware implementation of regular expression matchers is presented by Sidhu and Prasanna [163] for FPGA technology. This approach is also based on the McNaughton-Yamada construction of NFAs from REs. The surprising theme in their work is that the actual NFA construction can be performed in hardware, and the Self-Reconfigurable Gate Array (SRGA) can be automatically reconfigured in real time to pattern-match a new expression. Many comparisons are performed in the experimental results, one of which is a comparison to software-based pattern matching with DFAs. Software pattern matching often makes use of DFAs so that a single state can be tracked more easily, with the disadvantage that the DFA may be exponentially larger then its NFA counterpart. In hardware, simultaneously tracking multiple states is inherently done by the parallel circuitry, thus the smaller NFAs are often preferred. FPGA-accelerated pattern matching is used [165] to perform efficient network intrusion detection based on regular expressions.

Both instances of previous work [73, 163] have in common that NFAs are implemented in hardware to performs RE matching, and that $\epsilon$ transitions can be handled. The intersection and complementation operators in regular expressions are not treated. The work by Sidhu and Prasanna [163] is perhaps the most influential for the translation to circuits of the automata developed in the next chapter.

Other research by by Gheorghita and Grigore [82, 83] deals with the translation of PSL into automata. SERE intersection and fusion are treated, and their fusion algorithm [83] has similarities to the one developed in this work in Chapter 5. The details for the implementation of property operators are not given, with the exception of suffix implication [83]. Although a separate algorithm is used to implement suffix implication, in Chapter 5 it will be shown that suffix implication does not require a separate algorithm and can actually be implemented using the automata fusion algorithm. Moreover, in the most detailed reference [82], only SERE intersection is explained in algorithmic form. Although Boolean expressions are used on the edges of the automata, the implications for conventional automata theory are not developed. The minimization technique that is used consists of a set of ad-hoc rules, and no link is made to conventional DFA minimization. Experimental results show that the automata that are produced have more states compared to the checkers generated by the FoCs tool, which is presented further in this section.

PSL has been modeled in higher order logic for the HOL theorem prover by Gordon *et. al.* [88]. Once the semantics of PSL are captured in this formal reasoning tool, various proofs and "sanity checking" experiments can be make. In this framework, the semantics of PSL is said to be executable. HOL can also be used to produce a DFA from a PSL expression. The DFA can then be used to process a simulation trace in HOL to determine whether a given PSL assertion holds. In another application [88], a DFA can be converted to HDL thereby producing an assertion checker.

The automata produced in the work by Gheorghita and Grigore [83] and Gordon [88] can be used to check a property during simulation. These types of checkers indicate the status of the property at the end of simulation only, and are not ideal for debugging purposes. It is much more informative to provide a dynamic trace of the assertion and to signal each assertion failure: having a choice of violations to explore eases the debugging process, as some errors may reveal more than others. The time required to generate checkers is larger [83], sometimes significantly so [88] than with FoCs.

The tool that compares the most to the one developed in this thesis is the FoCs checker generator from IBM [1, 58, 108]. Since FoCs is a commercial tool, very few publications disclose its inner-workings. One particular characteristic that can be deduced from the literature [58, 106] is that FoCs also employs automata to generate HDL checkers from PSL assertions. The checkers produced by FoCs utilize an end-of-simulation signal which marks the end of time when strong properties are used. This signal is supplied by the user, and is used by the checkers so that any unfulfilled obligations can be reported as errors when no further cycles will be run. The end-of-simulation signal is also used in this work in Chapter 5, and is referred-to as an End-Of-Execution (EOE) signal.

At the time of this writing, the current version of FoCs is 2.04. FoCs does not currently support all property operators, and supports very few strong properties. The abort keyword is also not fully supported. In Chapter 7, the checkers produced by FoCs are compared to those produced by the tool developed in this work. The comparison involves generating checkers for a suite of assertions, and then synthesizing the checkers using FPGA implementation tools. The circuit size of the checkers is then compared using the number of flip-flops and combinational logic cells as metrics. Dating back to the early version of the checker generator [29], benchmarking against the FoCs tool has been a driving factor that has actually led to improvements in both tools. The results in Chapter 7 will show that the checker generator developed in this work outperforms FoCs.

In the PROSYD work [106], it is stated that the algorithms contained in that document are based on the algorithms implemented in the FoCs tool. The PSL algorithms are introduced in the context of generating checkers for simulation. The conversion of an NFA to a Discrete Transition System (DTS) is presented as the central result. A DTS is a symbolic program that represents an NFA, and is used during simulation for performing the assertion monitoring. The conversion of PSL assertions to NFAs is not developed, and only references other related model checking work [17]. The automata therein are developed for model checking and in the PROSYD document [106] it is not stated how they are adapted for use in dynamic verification.

In the IBM technical report [17] that serves as the basis for the PSL to NFA conversion used in the PROSYD project [106], two important issues are not treated: length-matching intersection of SEREs, and the use of a sequence directly as a property (as the right-hand side of temporal implication, for example). An important characteristic of the automata used is that Boolean expressions are used to form a non-mutually exclusive symbol alphabet, a theme that is central to the automata defined in the next chapter.

Other research was conducted by Gascard [79] on the transformation of SEREs to DFAs. This work is based on derivatives of regular expressions introduced by Brzozowski [38]. The derivative of a regular expression is a way of removing a given prefix in the language described by the regular expression. The result is also a regular expression. When applied repeatedly, this technique can be used to create a DFA from a regular expression. As will be shown in the next chapter, derivatives are not required to transform SEREs into DFAs. For one, algorithms will be devised to transform SEREs directly to NFAs that can subsequently be determinized when needed. Second, producing DFAs is not a prerequisite for RTL implementations since it was shown that NFAs are perfectly suitable for a circuit-form implementation [163].

It is a common solution to create a symbol encoding which represents the power set of the Boolean primitives, such that one and only one symbol is received at each step during the matching [79, 120]. This is referred to as the power set alphabet in the next chapter, and allows conventional automata theory to be used over Booleans. The power set alphabet will not be used; however, it is formally defined in the next chapter so that it can be compared to the symbolic alphabet. In essence, the power set alphabet maps all the possible valuations of the Booleans to distinct, mutually exclusive symbols, akin to the symbols in pattern and string matching. As will be shown, the disadvantage with the power set method is that an exponential number of edges are required to model a transition for a give Boolean. No benchmarks are performed for the SERE-to-DFA work in Gascard's publication [79].

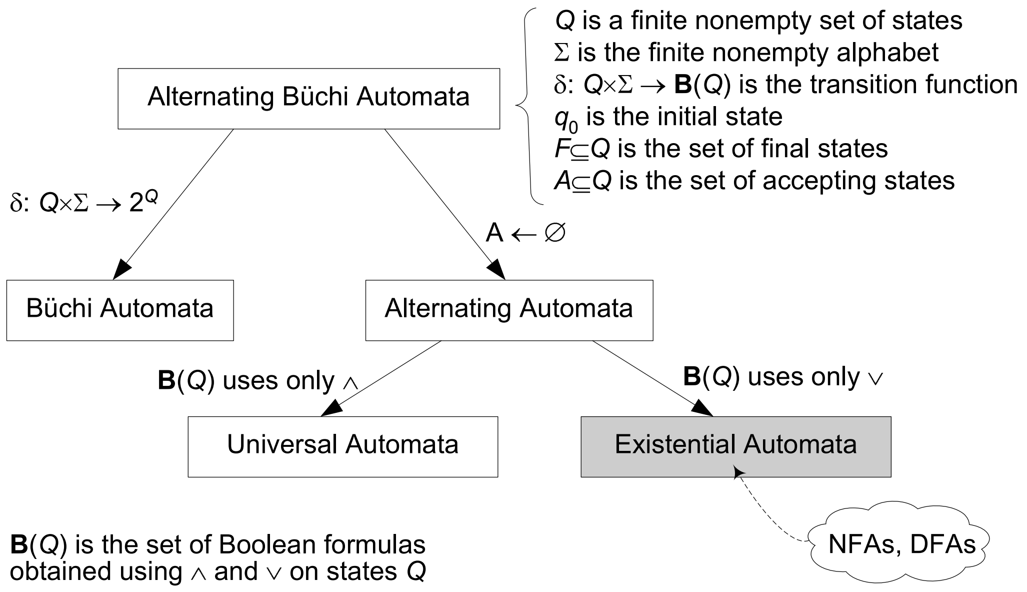

Q is a finite nonempty set of states
Σ is the finite nonempty alphabet
δ: Q×Σ → B(Q) is the transition function
q0 is the initial state
F⊆Q is the set of final states
A⊆Q is the set of accepting states
Alternating Büchi Automata
δ: Q×Σ → 2^Q
A ← ∅
Büchi Automata
Alternating Automata
B(Q) uses only ∧
B(Q) uses only ∨
Universal Automata
Existential Automata
NFAs, DFAs
B(Q) is the set of Boolean formulas obtained using ∧ and ∨ on states Q

**Figure 3.1:** Relations between the types of automata used in formal methods and conventional NFAs/DFAs.

## 3.3 Automata in Model Checking

Various types of automata have been used to implement PSL in the context of model checking [52]. Figure 3.1 shows how these different types of automata are related, and where the conventional NFAs and DFAs intervene. The definition of alternating Büchi automata is adapted from a document related to the PROSYD project [169]. The function $\mathbf{B}(Q)$ represents the set of all Boolean functions obtained by using conjunction ($\wedge$) and disjunction ($\vee$) on the set of states $Q$. At the top of the hierarchy is the *alternating Büchi automaton*, which defines the transition function in the most general manner using $\mathbf{B}(Q)$. Büchi automata accept omega-regular languages and are used in the context of infinite words, which differs from the behavior required for dynamic verification. The automaton defines a set of accepting states *and* a set of final states. An accepting state (as opposed to a final state) must be visited infinitely often in order for the automaton to accept the infinite run. Büchi automata are often used in automata construction for PSL model checking [40, 49, 169, 181]. Extensions to run-time property monitoring are also explored through the notion of testers [181].

As shown in Figure 3.1, *Büchi automata* are a particular case of alternating Büchi automata where the transition function maps only to subsets of states in the automaton (as opposed to Boolean functions of states). *Alternating automata* [174] are also a particular type of alternating Büchi automata, and are suitable for finite traces. Alternating automata do not define a set of accepting states, only the set of final states is used. *Universal automata* are a particular type of alternating automata in which only Boolean *conjunction* is allowed in $\mathbf{B}(Q)$. Universal automata are also used in formal property verification [19, 153].

*Existential automata* are another particular type of alternating automata, and in this case only Boolean *disjunction* is allowed in $\mathbf{B}(Q)$. The NFAs and DFAs introduced in the previous chapter fall into this category, as do the automata for assertions developed in the next chapter. In the transition function $Q \times \Sigma \rightarrow 2^Q$ (shown in Section 2.3), the disjunction is implicit between the members in the set of states that appear in the range of the function.

Alternating automata are used in the Sugar translator developed by Kargl [118], and in the VIS tool [170]. These types of automata are not suitable for a direct implementation in hardware since they must be de-universalized. Furthermore, although such procedures exist for converting alternating automata into existential automata, the run-time semantics exhibited by the automata developed for model checking are generally not suitable for creating checkers for run-time verification.

One aspect of some of the automata used in model checking [19, 153] that is also very relevant to the automata developed in the next chapter, concerns the nuances between semantic and syntactic alphabets. In conventional REs and languages, the symbol alphabet consists of a set of mutually exclusive tokens, and can be referred to as a syntactic alphabet. In SEREs, the syntactic elements are Boolean expressions whose truth values are independent and hence not mutually exclusive. In these cases, the *semantic* alphabet is different than the syntactic alphabet, and automata algorithms must take this into account. In automata that are based on semantic alphabets, determinization must take into account the fact that different symbols can simultaneously be true (as mentioned by Ruah, Fisman and Ben-David [153], without the explicit construction algorithm). In the next chapter, semantic alphabets are also called *symbolic alphabets*.

The notion of vacuity in model checking is treated by Kupferman and Vardi [123]. An informal definition of vacuity is given next, and applies to both static and dynamic verification.

**Definition 3.1:** A property implication is vacuously true (a *vacuous success*) when the conditions in its antecedent are never satisfied.

In static verification, if an antecedent condition can never be true in a model, this indicates either a problem in the model or that the property is irrelevant. In dynamic verification, if an antecedent condition never occurs, this indicates either that the test scenario is not exercising the design sufficiently, or that the property is also irrelevant. Identifying vacuity in circuit-level assertion checkers can be accomplished using two debug enhancements that will be presented in Section 6.4.

The notion of vacuous success is associated to properties that make use of property implications. The notion of *trivial validity* is developed for model checking by Beer *et. al.* [10], and extends the definition of vacuity to cover other cases where properties are trivially true. Examples of vacuity and trivial validity are shown next, with their reasoning.

**Example 3.1:** The following properties are assumed to be relevant to the design being verified:

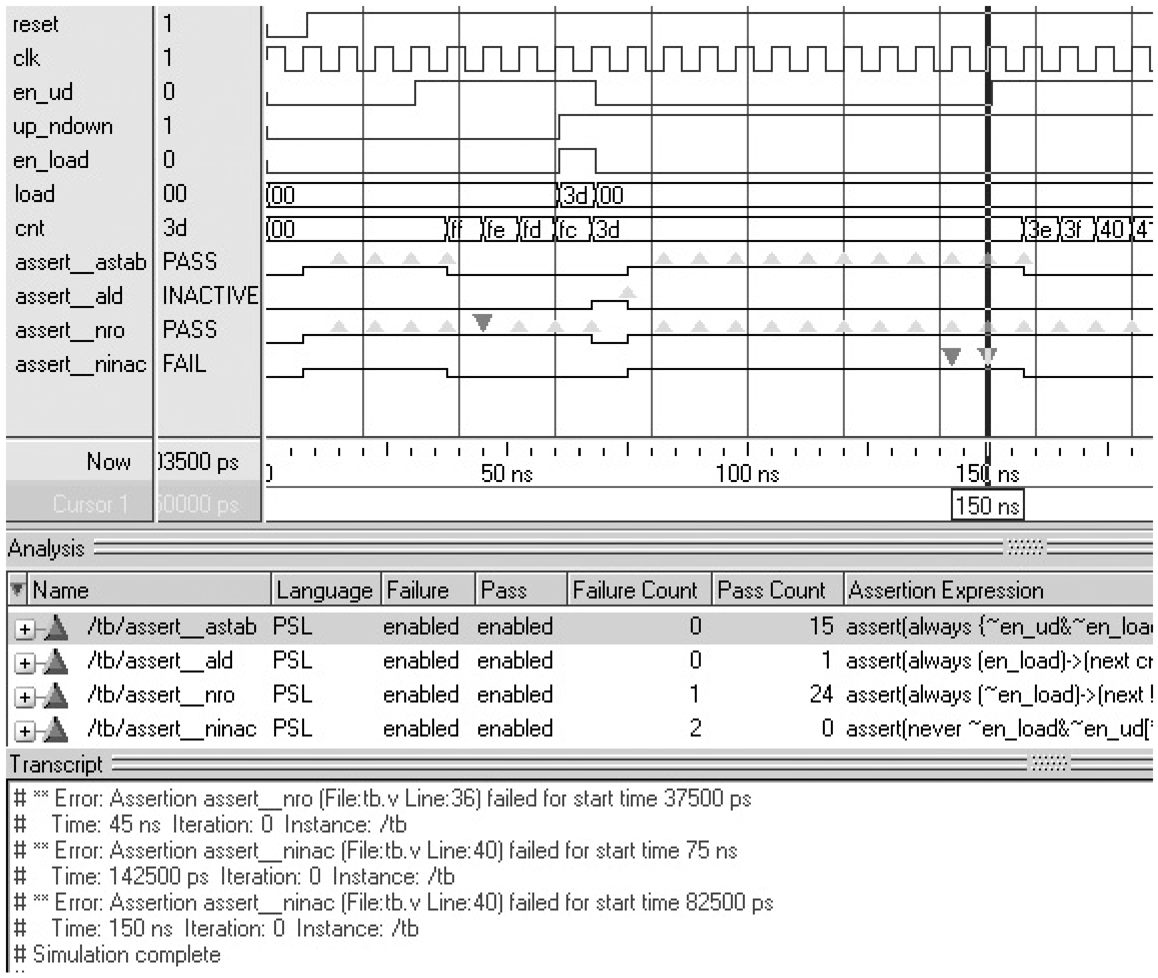
$$\mathsf{always}\ \{a\&b\}\ |\texttt{=>}\ (p\ \mathsf{until}\ c)$$
$$\mathsf{never}\ \{\{[*1]\}\ \&\&\ \{[*2]\}\}$$

The first property is used to exemplify vacuous success and trivial validity. If signals $a$ and $b$ can never simultaneously be true in the model, then the entire property holds vacuously and is of no use. In simulation, if $a$ and $b$ are never simultaneously true then the effectiveness (and coverage) of the testbench can be questioned. If $a$ and $b$ do occur simultaneously, but $c$ always occurs in the next cycle then the sub-property $p$ is never checked and is deemed trivially valid. The second property is a tautology because the length matching intersection of two different-length SEREs can never hold, and is also trivially valid.

A common theme in the automata used in the next two chapters and much of the referenced work [17, 19, 106, 118, 153, 170] is that Boolean expressions are used on automata edges. As will be shown, one aspect where the current work diverges from the previous work is in the encoding of symbols for automata. Furthermore, the checkers produced by the algorithms in this thesis exhibit a run-time assertion-monitoring behavior that is well suited for the hardware execution of checkers.

## 3.4 Assertion Support in Simulators and Emulators

When assertion-based verification is to be performed dynamically, HDL simulators capable of interpreting PSL are often used. Examples of such tools are Synopsys VCS, Cadence's Incisive Unified Simulator and Mentor Graphics' ModelSim. The main task of the simulator is to process the assertions that are either embedded in



| | |
|---|---|
| reset | 1 |
| clk | 1 |
| en_ud | 0 |
| up_ndown | 1 |
| en_load | 0 |
| load | 00 |
| cnt | 3d |
| assert__astab | PASS |
| assert__ald | INACTIVE |
| assert__nro | PASS |
| assert__ninac | FAIL |
| Now | )3500 ps |
| Cursor 1 | )0000 ps |

Analysis

| Name | Language | Failure | Pass | Failure Count | Pass Count | Assertion Expression |
|---|---|---|---|---|---|---|
| /tb/assert__astab | PSL | enabled | enabled | 0 | 15 | assert(always {~en_ud&~en_loa |
| /tb/assert__ald | PSL | enabled | enabled | 0 | 1 | assert(always (en_load)->(next cr |
| /tb/assert__nro | PSL | enabled | enabled | 1 | 24 | assert(always (~en_load)->(next ! |
| /tb/assert__ninac | PSL | enabled | enabled | 2 | 0 | assert(never ~en_load&~en_ud[ |

Transcript

```
# ** Error: Assertion assert__nro (File:tb.v Line:36) failed for start time 37500 ps
#    Time: 45 ns  Iteration: 0  Instance: /tb
# ** Error: Assertion assert__ninac (File:tb.v Line:40) failed for start time 75 ns
#    Time: 142500 ps  Iteration: 0  Instance: /tb
# ** Error: Assertion assert__ninac (File:tb.v Line:40) failed for start time 82500 ps
#    Time: 150 ns  Iteration: 0  Instance: /tb
# Simulation complete
```

**Figure 3.2:** Example of assertions interpreted by simulators for dynamic verification.

the source code of the design or specified in a separate file, in order to monitor the simulation and to report any circuit behavior that violates the assertions.

An example simulation with ModelSim SE version 6.1f is shown in Figure 3.2 for the assertions in Example 2.7. The assertions are interpreted by the simulator during the execution of the testbench. The testbench instantiates the design under verification, which in this case is the up-down counter from Example 2.7. In Figure 3.2, the simulation shows that the two "environmental" assertions failed: the environment did not respect the inactivity and rollover criteria established (the third and fourth assertions in the example). The first two intrinsic-functionality assertions did not fail; however, to increase the confidence in the design, more test cases should be executed to improve the coverage of the dynamic verification.

The wave section at the top of the figure shows the time points where the assertions failed (downward-pointing triangles), and the transcript at the bottom shows for which start times the assertions actually failed. The analysis pane in the center shows the kind of supplemental information that can be provided by simulators. The tool is able to report the number of times each assertion failed, and is also able to report the number of times each assertion completed successfully. Successful completions are indicated by the lighter upward-pointing triangles in the waveforms. Assertion completion and assertion counters are two of the debug enhancements that are also implemented in the hardware checkers in this work, and will be introduced in Section 6.4.

To continue the discussion on run-time semantics from Section 2.2, it should be emphasized that the PSL specification does not dictate how and when the assertion failures are to be reported in dynamic verification. Some assertions interpreted by simulators only report one failure for a given start condition. For example, in the following assertion,

$$\textsf{assert never } \{a;b[*0{:}1]\};$$

for every cycle in which $a$ is asserted an error will be reported. However, if the cycle that follows $a$ has $b$ asserted, in ModelSim a supplemental error will not be reported. This is perfectly acceptable given that the run-time semantics of PSL is not specified. If an assertion fails at one or more time points, it has failed globally. One possible reason for this behavior in ModelSim is that in the simulation kernel, the threads that monitor assertions are kept as short as possible for performance reasons. In hardware, this is not a concern because the assertion is implemented as a circuit. In the checker generator developed in this work, a failure is reported when $a$ is observed *and* when $a\,;b$ is observed. In other words, in software it is more computationally efficient to stop monitoring the thread when $a$ has occurred, and in hardware it is more resource efficient to let the pattern matching circuit follow its regular flow.

As was just observed, interpreting assertions in software is very different than in hardware. In software approaches, one can take advantage of features such as stack-based function calls, recursive functions, threads and event lists, and most importantly, dynamic memory allocation. An example of software-based PSL interpretation is published by Chang *et. al.* [45] where threads and event lists play a central role.

When excessive simulation time becomes a bottleneck for dynamic verification, hardware emulation and simulation acceleration are often used. In order for the ABV methodology to be used in hardware emulation, assertions must be supported in hardware. Traditional emulators are based on reconfigurable logic and FPGAs. To increase flexibility and to ease the debugging process, which requires the ability to instrument assertions, current-generation emulators and simulation accelerators are typically based on an array of processing elements. The Cadence Palladium [42] and the Tharas Hammer [78] are examples of such emulators and simulation accelerators. Mentor Graphics also recently introduced the Veloce accelerator and emulator, with support for transaction level verification [130]. These tools support the use of assertions.

The ZeBu FPGA-based emulator family from Eve supports the use of SystemVerilog assertions in hardware in a variety of emulation products [71]. A commonality with the simulators and emulators described in this section is that since they are all commercial products, the internal details of how PSL is implemented are not published.

## 3.5 Assertion Checkers in Silicon Debugging

In the emerging Design for Debug (DFD) paradigm, several EDA companies are promoting a variety of solutions. Tools from companies such as Novas support advanced debugging methods to help find the root cause(s) of errors by back-tracing assertion failures in the RTL code [102, 167]. Temento's DiaLite product accepts assertions and provides in-circuit FPGA debugging features. Synopsys' Identify Pro allows assertions to be synthesized into hardware (FPGA or ASIC), and for these assertions to be used as a triggering mechanism for capturing the state of the design when failures occur. However, as these tools are from commercial ventures, papers seldom disclose their actual inner-working.

As increasing transistor counts and smaller process technologies make it difficult to achieve correct silicon, techniques for post-fabrication debugging, known as silicon debugging, are receiving much attention. DAFCA's ClearBlue solution [2] offers silicon debugging instruments such as signal probe multiplexers, logic analyzer circuitry, and in-circuit trace buffers for capturing signals or supplying test vectors. To ensure flexibility in providing these post-silicon debug instruments, they are implemented in small blocks of additional programmable logic. Assertions can also be instrumented and changed dynamically in the specialized reprogrammable logic cores. The status of the RTL silicon-debug instrumentation that is added to the source design can be read back through the JTAG interface. Special debug circuitry and read-back is also part of the research on assertion-based debugging presented by Peterson and Savaria [149].

When many assertion checkers are to be used in a modestly-sized programmable logic core, within a System-on-Chip for example, checkers must be managed in groups. The idea of assertion grouping is also mentioned by Abramovici *et. al.* [2], and is explored further in Section 6.5. The debugging instruments also introduced by Abramovici *et. al.* [2] and the checker enhancements presented in Section 6.4 encompass a collection of techniques that share a common goal: to help increase the efficiency of the debugging process with assertions.

Recent work has shown that specifying various types of transfer sequences, phases and corner cases in a hierarchical and higher-level manner can be used to more efficiently automate the generation of protocol monitors [137]. Graphical user interfaces are also shown to play a key role in facilitating the debugging process with assertions, and can even be used in the specification of test sequences.

Cross-product functional coverage is explored by Ziv [182] to reduce the amount of assertions that need to be specified for verification. Auxiliary variables are assigned to sub-expressions and are incremented when these sub-expressions are executed. In this way, various combinations of expressions can be reported and a much larger coverage space can be measured. However, the statement made in that work that is the most relevant to the checker generator concerns the semantics of the cover operator in PSL. It is stated that expressing a coverage task for an expression $e$ is equivalent to asserting (in LTL notation): $F\ e$. In PSL, the previous statement implies that covering a sequence is equivalent to asserting that it must *eventually* be observed. This is the basis of the rewrite rule that will be used in Section 5.5 to implement the cover directive.

## 3.6 Other Related Research

Given the parallels between conventional REs and SEREs over Booleans, the vast literature on converting regular expressions into automata is summarized through a few key references. Berry and Sethi [21] show how to build DFAs from REs using the notion of derivatives of regular expressions [38]. In Raymond's work [152], although the object of the RE transformations are not automata but rather Boolean dataflow networks, the modularity of the approach has parallels to the modular approaches discussed in Section 3.1. The McNaughton-Yamada construction [128] shows how to produce an NFAs containing $\epsilon$ transitions from REs. The first four chapters in the Hopcroft textbook [101] also present automata, regular expressions and languages, and is the classic reference used for when these topics are introduced to students.

Generating monitor circuits from Generalized Symbolic Trajectory Evaluation

(GSTE) specifications is researched by Hu *et. al.* [103, 139]. GSTE specifications are represented using assertion graphs, and consist of automata with antecedent and consequent symbols on the edges. Symbolic constants can also be used to allow more general and powerful properties to be specified. The monitor construction is based on a modular assembly of sub-circuits corresponding to the structure of the assertion graph. A token-based approach is also used, and is evocative of the token-based implementation of SEREs described in Section 3.1. An important topic covered by Hu *et. al.* is the development of a simulation-friendly specification for GSTE to allow efficient monitor circuits to be built. This is not unlike the simple subset restrictions defined for PSL in order for assertions to be suitable for dynamic verification.

Production Based Specification (PBS) developed by Seawright and Brewer, advocates the use syntax productions to perform actual circuit design [158, 159, 160]. The language used has many similarities to SEREs, however certain operators do not behave the same way as in PSL. The sequential-and operator consists in performing the Boolean conjunction of the *result* of sequences (i.e. in their final cycle), and does not equate to the length-matching intersection found in assertion languages. Furthermore, the sequential-not operator does not correspond to the type of negation required to perform sequence-failure detection, as the negation is performed on the *result* signal of a sequence. An interesting statement by Seawright [160] reveals that using Boolean expressions as tokens allows an efficient *symbolic alphabet* to be used. Experimental results include the high-level design of a mouse decoder circuit.

As its name implies, the SystemVerilog Assertions (SVA) language is an assertion language that is part of SystemVerilog [110]. Many similarities exist between SVA and PSL assertions [96], most notable of which are sequences. Interestingly, SVA defines an operator called first_match that is used to report the first match of a sequence. Although this operator has no direct equivalent in PSL, a corresponding algorithm will be devised in Section 6.4 to implement the *completion-mode* debugging enhancement. Contrary to the sequences in both languages, SVA properties are not as similar to PSL properties. Although SVA properties define suffix implication and an abort operator, LTL constructs are not currently available.

One advantage with SVA assertions is that an action block allows arbitrary HDL code to be executed when a property fails. Currently, there is no provision for this in the PSL specification; however, using the circuit-level checkers to interpret assertions can be a way around this problem. In checkers, assertion signals are regular HDL signals and they can be used for any purpose, including providing feedback to a testbench to guide the stimulus generation in a closed-loop test scenario.

Open Vera Assertions (OVA) is an open source assertion language based on LTL and regular expressions [166]. The language also provides feature to reuse libraries of pre-built assertions that can be shared across the verification community. The ForSpec language developed by Intel [6] is also based on regular expressions and linear temporal logic, and was actually a candidate along with IBM's Sugar language for the first version of PSL that was developed by Accellera (incidentally, it was IBM's language that was selected). Both the OVA and ForSpec languages are suitable for formal and dynamic verification. Many automata-based implementations of LTL are also developed for formal verification, where properties in finite LTL can be implemented in automaton form for simulation purposes [155].

In certain scenarios it may be preferable to avoid using assertion languages altogether, and instead rely on a pre-compiled set of checkers. One such library is the Open Verification Library (OVL) [75]. In this methodology, assertion checkers are simply instantiated in the design under verification, and connected to the appropriate design signals. The types of properties that can be used range from generic "always"-type properties to complex bus protocol monitors. Some EDA vendors also have proprietary verification IP (Intellectual Property), such as Mentor's CheckerWare verification IP library. The disadvantage of these libraries is that pre-defined components may not always be found for certain specialized applications. The checker generator developed in this work can be an ideal way to build and maintain a library of pre-compiled checkers.

The e-Language [109] is a programming language used to assist in the verification of hardware designs. It is used to create complex testbenches and functional verification environments for stimulating and analyzing a design's behavior. Many high-level features allow coverage driven verification and constrained random verification to be performed. The language is built upon a variant of linear temporal logic, and also defines temporal expressions that are somewhat similar to PSL sequences.

Hardware monitors are somewhat analogous to the observers developed for monitoring properties of distributed software, as modeled in SDL (Specification and Design Language) for example [93]. Generating monitors from visual specifications and timing diagrams has also been explored [77, 141]. Research is also being performed to incorporate assertions into the SystemC modeling environment [58, 89, 90]. Transaction-level assertions [65] and system level assertions [126] are also being developed but are outside the scope of this work.

# Chapter 4

# Automata for Assertion Checkers

## 4.1 Introduction and Overview

The goal of Chapters 4 and 5 is to develop the methods for generating circuit-level assertion checkers from high-level assertion statements. This chapter introduces the automaton framework used for implementing assertion checkers, while the next chapter explains how automata are built for the various syntactic elements of PSL, ranging from Booleans to full verification directives.

Utilizing classical automata theory directly for pattern matching over Boolean propositions implies a power set symbol mapping that can impose a high penalty on the number of edges required in the automata. For this reason, a symbolic alphabet will be developed and used as the alphabet. This is not without consequence however, as many of the conventional automata algorithms can not be used with a symbolic alphabet. The problem is that conventional RE pattern matching is defined upon a mutually exclusive symbol alphabet, where one and only one symbol is received at each step. This is not the case with Boolean expressions, which can simultaneously and independently evaluate to true. Modifications to well established automata algorithms will be a recurring theme in Chapters 4 and 5, for operators that are common both to REs and SEREs. For other operators that are not used in SEREs, special algorithms will be devised, also taking under consideration the symbolic alphabet. It is also at the end of the present chapter that the conversion of automata to RTL circuits is performed.

## 4.2 Automaton Framework

In this section, the automaton framework used for creating assertion checkers is formally defined. Of primary importance is the introduction of a dual layer symbolic alphabet, and its effects on conventional automata algorithms such as determinization and minimization. The underlying goal is to generate smaller automata, so that when expressed as a circuit, the generated assertion checkers utilize fewer hardware resources, and are implemented less obtrusively in FPGAs or ASIC hardware.

### 4.2.1 Automaton Definition and the Symbol Alphabet

Contrary to the automata for conventional regular expressions presented in Definition 2.2, the automata for assertion checkers are not defined over a mutually exclusive alphabet. In order to use the defined automaton formalism for the case of Booleans, the power set alphabet is traditionally used [120].

**Definition 4.1:** The *power set* alphabet corresponds to all the possible valuations of the Booleans used in an assertion. If there are $B$ Booleans, it follows that the cardinality of the alphabet is $|\Sigma| = 2^{|B|}$.

When an assertion references many Booleans, the exponential increase in the numbers of symbols and edges make this approach difficult to scale. The advantage of the power set alphabet is that symbols become mutually exclusive, and conventional automata algorithms can be used directly; the framework becomes analogous to conventional string matching where one and only one character is received at a time. In sum, the power set alphabet brings the non-mutually-exclusive Booleans into the realm of the mutually exclusive symbols of traditional pattern matching. The disadvantage is that an exponential number of edges are required to model a transition.

To avoid the exponential number of edges required in the power set alphabet, a symbolic alphabet is developed.

**Definition 4.2:** The *symbolic alphabet* corresponds to a direct mapping of the Booleans used in an assertion, such that each symbol represents a given Boolean directly.

Although simple in appearance, this alphabet is *not* mutually exclusive and requires special modifications to algorithms such as intersection and determinization. In the symbolic alphabet, edge symbols represent complete Boolean-layer expressions that are not mutually exclusive because any number of separate expressions can *si-*

**Table 4.1:** Power set and symbolic alphabets for Example 4.1.

| $c$ | $b$ | $a$ | **Power set alphabet** |
|---|---|---|---|
| F | F | F | $\sigma_0$ |
| F | F | T | $\sigma_1$ |
| F | T | F | $\sigma_2$ |
| F | T | T | $\sigma_3$ |
| T | F | F | $\sigma_4$ |
| T | F | T | $\sigma_5$ |
| T | T | F | $\sigma_6$ |
| T | T | T | $\sigma_7$ |

| **Symbolic alphabet** |
|---|
| $\sigma_a = a$ |
| $\sigma_b = b$ |
| $\sigma_c = c$ |
| $\sigma_{b \wedge c} = b \wedge c$ |

*multaneously* evaluate to true. This creates nondeterminism because a given state may transition to more than one state. While adding inherent nondeterminism in this alphabet, certain determinizations throughout the operations on automata can be avoided, which helps create smaller automata.

The alphabet stated above is only partially complete, and may grow as new symbols are required. The following example based on sequence fusion illustrates the fundamental differences between the two alphabet approaches, and the effects on the number of edges required.

**Example 4.1:** The fusion of two sequences defined from Booleans $a$, $b$ and $c$ follows:

$$\{a;b\}:\{c;a\}$$

Table 4.1 shows how the power set alphabet is defined for this example. The symbolic alphabet is also shown; however, before the fusion is actually performed, only the symbols $\sigma_a$, $\sigma_b$ and $\sigma_c$ exist. In the table $\wedge$ is Boolean conjunction, $\vee$ is disjunction, T is short for `true` and F is `false`. In the power set alphabet (left), only one line (symbol) is active at any given time, thus four symbols (and edges) are required to model a transition for a given Boolean; in the symbolic alphabet, only one symbol is required. For example, to model a transition on Boolean $b$ in the power set alphabet, four edges are required for symbols $\sigma_2$, $\sigma_3$, $\sigma_6$ and $\sigma_7$.

Figure 4.1 illustrates the effect of the choice of symbol alphabets on the automata for the current example. Although the fusion algorithm is presented in the next chapter, the emphasis here is more on the number of edges required than the actual fusion operator.

The symbolic approach is more efficient in terms of the number of edges required,

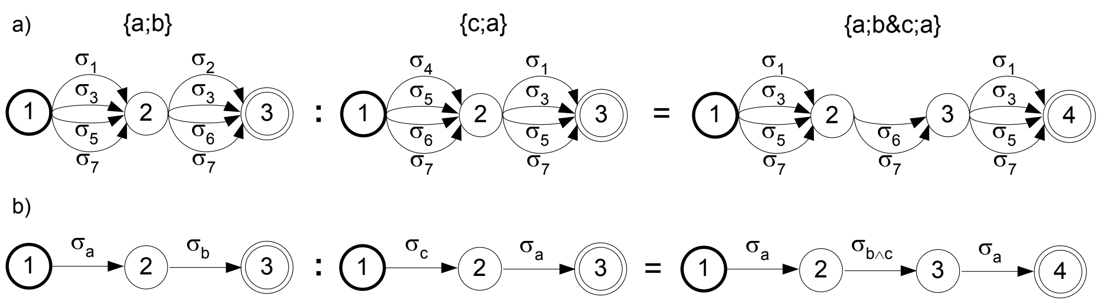


**Figure 4.1:** Effect of alphabet choice on automata for Example 4.1 using: a) power set alphabet, and b) symbolic alphabet.

and the number of states is unaffected. In general, when $n$ Booleans are used in an assertion, a transition on a given Boolean actually requires $2^{n-1}$ edges, whereas only one edge is used in the symbolic alphabet. The symbolic alphabet developed in this work is actually based on a two-layer symbolic alphabet, using primary and extended symbols. Henceforth the expression *symbolic alphabet* will refer to the actual two layer alphabet described below as part of the formal automaton definition.

**Definition 4.3:** In this work, a *Finite Automaton* $\mathcal{A}$ is described by the six-tuple $\mathcal{A} = (Q, \Pi, \Sigma, \delta, I, F)$, where:

- $Q$ is a nonempty finite set of states;
- $\Pi$ is a set of primary symbols that represent Booleans;
- $\Sigma$ is a set of extended symbols defined from $\Pi$;
- $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation consisting of a subset of triples from $\{(s, \sigma, d) \mid s \in Q,\ \sigma \in \Sigma,\ d \in Q\}$;
- $I \subseteq Q$ is a nonempty set of initial states;
- $F \subseteq Q$ is a set of final (or accepting) states.

In contrast, the conventional automaton in Definition 2.2 has a single initial state (as opposed to a set of initial states); the conventional automaton also allows the use of $\epsilon$ transitions, and has a mutually exclusive symbol alphabet, not to mention a single-level alphabet. In Definition 4.3 above, the alphabet is defined by the $\Pi$ and $\Sigma$ sets, which represent non-mutually-exclusive Booleans. The $\Pi$ and $\Sigma$ sets (defined below) are global symbols that are shared across multiple automata. Henceforth the shortened notation $\mathcal{A} = (Q, \delta, I, F)$ will be used.

The term *edge* is used to refer to an element of the transition relation $\delta$, and is represented by an ordered triple $(s, \sigma, d)$, where an edge's extended symbol is denoted

$\sigma$, and $s$ and $d$ are used to denote the source and destination states of an edge respectively. The transition relation is not *complete*, and there does not have to be a transition for each symbol in each state.

Each state also has a *label*, consisting of either an ordered pair, or a totally ordered set, depending on the application. Labels are an essential part of some of the algorithms appearing further. In the automaton definition for assertion checkers over Booleans, the language accepted by the automaton actually represents a set of traces. This set of traces violates the behavior given by the assertion, and it is the automaton's purpose to report such traces.

The notation $\mathcal{A}(x)$ is used to denote the construction of an automaton from an argument expression $x$, where $x$ can range from a simple Boolean to a full verification directive based on properties and sequences. The Booleans appearing in sequences and properties are used to form the primary symbols in the automaton's symbol alphabet, and primary symbols are then used to construct the actual extended symbols that appear on the edges in the automata.

**Definition 4.4:** A *primary symbol* $\pi$ represents the HDL code of a Boolean expression appearing in a sequence or property, with any outermost negation(s) removed. The set of primary symbols is denoted $\Pi$.

**Definition 4.5:** An *extended symbol* $\sigma$ is a single literal, or a conjunction of multiple literals, where a *literal* is a negated or non-negated primary symbol. A literal has a negative polarity if and only if its corresponding primary symbol has an odd number of outermost negations removed when entered as a primary symbol. The set of extended symbols is denoted $\Sigma$.

The $\epsilon$ symbol from conventional regular expressions represents the empty match, and when used as an edge's symbol, $\epsilon$ transitions are in effect instantaneous transitions that do not require an execution step of the automaton. As a result of Definition 4.5 however, $\epsilon$ symbols and transitions are *not* allowed in the automaton framework for assertion checkers.

Disjunction of literals is not handled in the extended symbols because disjunction can be performed by two parallel edges that originate (and terminate) at the same pair of states. When the automaton is constructed for a given assertion, the Boolean primitive true may be added to the set of primary symbols, even when it is not used directly in an assertion. For example, the primitive $\pi =$ true is required for implementing the always property (as will be shown in an example in Section 4.3), and is eventually added to Π during the construction of the automaton.

**Example 4.2:** To show how Booleans and symbols interact, the following assertion based on a two cycle sequence and the never operator is used:

$$\textsf{assert never } \{\ req\ \&\ gnt\ ;\ \sim(ack\ |\ \sim gnt)\ \};$$

The primary and extended symbol sets built from this assertion are, respectively:

$$\Pi = \{\pi_1, \pi_2\} \quad \text{where: } \pi_1 = \text{“}req\ \&\ gnt\text{” and } \pi_2 = \text{“}(ack\ |\ \sim gnt)\text{”}$$
$$\Sigma = \{\sigma_1, \sigma_2\} \quad \text{where: } \sigma_1 = \pi_1 \text{ and } \sigma_2 = \neg\pi_2$$

The extended symbols in Example 4.2 require only single literals; however, for further processing in an automaton form, other symbols may be added, some possibly referencing a *set* of conjoined literals. The set of literals comprising an extended symbol is totally ordered, such that no duplicate symbols are kept for equivalent conjunctions that have a different ordering of literals. In the implementation, the total order relation is that which is produced by sorting the actual data elements that represent literals.

For convenience, most often when automata are illustrated the expressions represented by the extended symbols on each edge are constructed from the symbol tables and are shown above or below their corresponding edges, as opposed to showing extended symbol identifiers $\sigma_i$ themselves, which require that the sets Π and Σ be explicitly specified. This way an automaton can be understood quicker and the symbol alphabet tables do not need to be shown. Example 4.2 can also be used to illustrate how the dual-layer symbols can be assembled in reverse to construct the Boolean expressions of an assertion.

An extended symbol can be interpreted with a truth assignment $\omega$ over its referenced primary symbols. The truth value of an extended symbol $\sigma$, under *assignment* $\omega$, is denoted $(\sigma)_\omega$, and evaluates to either true or false. For example:

$$\sigma = \pi_1 \wedge \neg\pi_2$$
$$\omega \leftarrow [\pi_1 = \textsf{true}, \pi_2 = \textsf{true}]$$
$$(\sigma)_\omega = \textsf{false}$$

By definition, if $\sigma = \textsf{true}$ then $(\sigma)_\star = \textsf{true}$, where $\star$ represents any assignment possible.

In the symbolic alphabet, any given symbol $\sigma$ from Σ represents a Boolean expression. A transition on $\sigma$ takes place when the Boolean expression represented by $\sigma$ is true *and* the source state of the transition is active. The output signal of the assertion checker corresponds to a disjunction of all final states. When a final state is active in a given clock cycle, the pattern described by the automaton has been matched. In this light, constructing an automaton for assertions actually means constructing an automaton to detect all faulty traces that do not conform to the behavior dictated by an assertion.

The clock does not appear in the automaton definition, but is implicitly used to regulate the transition from one set of active states to another. This is analogous to conventional automata for string matching, where reading a character causes the automaton to transition. The definition of deterministic automata made in Definition 2.3 for conventional automata also applies to the symbolic automata defined in this section.

By extension, Corollary 2.4 states that conventional FAs are deterministic (DFA) when they have a single initial state and no more than one outgoing edge from a given state *carries the same symbol*. Since the Boolean expressions represented by automaton symbols are not mutually exclusive, the corollary does not work with the symbolic alphabet, and Definition 2.3 must be adhered to. This fundamental difference between both automata models influences many operations, not the least of which is the determinization function.

The symbolic automata introduced in this chapter do not have to be complete (Definition 2.5), and in a given state, it may be the case that no outgoing transitions are taken in a given cycle. As illustrated in Example 2.1, a complete DFA accepts the same language as a DFA. Since a complete DFA has more edges (and possibly more states) than a DFA, complete DFAs are avoided when possible.

It is also observed in this work that automata with multiple initial states can be more compact than automata that are restricted to having a single initial state. When automata are expressed in a circuit form, having a set of active states is perfectly acceptable. In such cases the initial states are the only active states when the reset is released. It is for this reason that the automaton definition has a *set* of initial states, as opposed to the automata with single initial states from Definition 2.2. Figure 4.2 shows an example whereby allowing a set of initial states produces an automaton with fewer edges.

The dual symbol alphabet allows the automaton to be "aware" of the polarity of signals and also of more Boolean simplifications when conjunctions of edge symbols occur. Since the intersection, fusion and determinization operations require the inter-

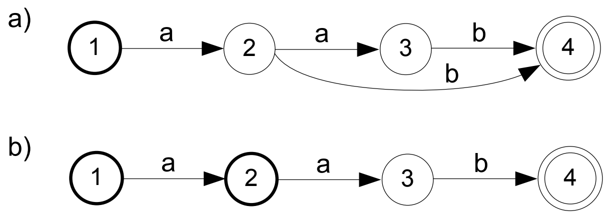


**Figure 4.2:** Effects of multiple initial states for the sequence $\{a[*1{:}2];b\}$ : a) restricted to a single initial state; b) *set* of initial states, thus requiring fewer edges.

section of symbols (and also polarity manipulations in determinization), new primary symbols do not have to be declared for a conjunction. For example, in Section 2.4 it was shown that a goto repetition $a$[->] is equivalent to $(\sim a)$[*];$a$. The following example builds on this case and illustrates why the two-layer symbol alphabet is beneficial for reducing the size of the automata. For now, understanding the rewrite rule used and the automaton construction from PSL statements is not required, and will be presented in the next chapter.

**Example 4.3:** The following circuit assertion stipulates that two given subsequences must never simultaneously occur, starting and terminating at the same time. The second assertion's SERE was modified by a rewrite rule, and is more representative of what is actually seen by the checker generator algorithms.

```
assert never {{a[->];b[*]} && {a;c;d}};
assert never {{{(~a)[*];a};b[*]} && {a;c;d}};
```

The rewritten assertion is expressed in automaton form as shown in Figure 4.3 a), using dual-layer symbols. The `never` property has the effect of creating the `true` self-loop in state 1. The automaton has four states, compared to six states when single-layer symbols are used. With only one level of symbols, a new symbol must be created for $(\sim a)$, and the automaton is not "aware" that $(\sim a)$ and $a$ are related, as can be seen in Figure 4.3 b) in the edge from state 1 to state 2.

The automaton approach to checker generation allows many types of optimizations to be performed. The following examples show different cases where a modular approach will generate more RTL checker code for the assertions on the left, even though in dynamic verification the assertions on the left are semantically equivalent

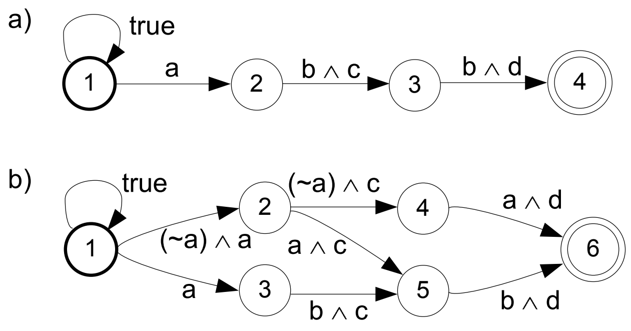


**Figure 4.3:** Automata for assertion in Example 4.3 using: a) dual-level symbols, b) single-level symbols.

to their counterparts on the right.

$$\text{assert always } \{a[\text{->}]\} \mid\text{=>}\ b; \quad \equiv \quad \text{assert always } \{a\} \mid\text{=>}\ b;$$
$$\text{assert never } \{b[{*}1{:}2];c\}; \quad \equiv \quad \text{assert never } \{b;c\};$$

In the first example above, in the assertion on the left, the goto matching related to $a$ is not required because the always continually retriggers the matching of its argument, thus the extension created by the goto operator is redundant. In the second example above, the never operator also continually triggers the matching of its argument, and in such cases the sequence $\{b; c\}$ is guaranteed to be matched when the sequence $\{b; b; c\}$ is matched, hence the first $b$ is redundant in the range repetition.

Another type of example appears next, which yields identical *five*-state automata in both cases, as opposed to the modular approach that would generate much larger code in the first case, where a link between the arguments of next_event_a is not made ($b$ vs. $\sim b$).

$$\text{assert always } a \text{ -> next\_event\_a}(b)[1{:}2](\{\{\sim b;c[{*}200];d\} | \{e;d\}\});$$
$$= \quad \text{assert always } a \text{ -> next\_event\_a}(b)[1{:}2](\{e;d\});$$

Although synthesis tools can perform the necessary optimizations to reduce circuits like the one in Figure 4.3 b), in general, not all sequential reductions can be performed this way. Notwithstanding the synthesis aspect, the automata should also be kept as small as possible because operations such as intersection and determinization have worst case behaviors that are product and exponential, respectively, in the number of states.

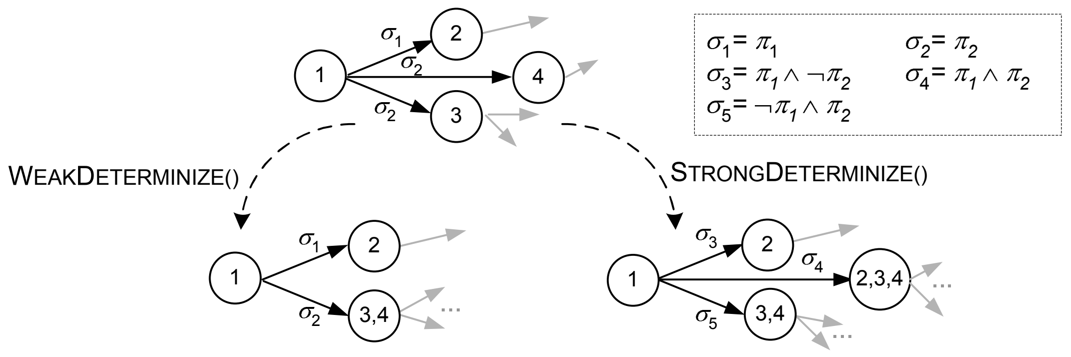


**Figure 4.4:** Strong vs. weak determinization.

## 4.2.2 Determinization of Automata

Classical determinization typically does not consider the possibility that edge symbols represent Boolean expressions, many of which can simultaneously be true. Usual determinization procedures [101] have the effect of creating an automaton with a single initial state, and for which no state has two outgoing transitions using the same symbol. In conventional automata, this is a sufficient criterion to ensure determinism (Corollary 2.4); however, this is not sufficient given the requirement for determinism that was made in Definition 2.3. Therefore, in this work the usual determinization algorithm is said to create *weakly deterministic* automata; although strictly speaking the automata produced may in fact not be deterministic. The nondeterminism that is left by weak determinization will be used advantageously in the minimization algorithm in the next subsection to help keep the automata as small as possible.

The effect of the WEAKDETERMINIZE() algorithm is first illustrated in Figure 4.4 using a small example. The top of the figure shows a portion of a NFA and the bottom left shows the weakly determinized version. In general, a new state in the DFA is labeled as "$i, j, k, \ldots$", where $i, j, k, \ldots$ are states of the input NFA. When the NFA at the top of the figure is weakly determinized, only the symbols appearing directly on the edges are considered. In the automaton at the bottom left, no two outgoing edges carry the same symbol. If the symbols were mutually exclusive, as in string matching for example, the weakly determinized automaton would be truly deterministic.

Some of the algorithms in the next chapter actually require proper determinization, which is referred to as *strong determinization*. The effect of this algorithm is also illustrated in Figure 4.4. In the strongly determinized automaton at the bottom right in the figure, no conditions allow state 1 to transition into more than one state. The strong determinization creates new symbols by manipulating polarities of

```
FUNCTION: WeakDeterminize($\mathcal{A}$)
//a label is a totally ordered set $T \subseteq Q$
**apply** total order to $I$
**create** new state $q$ labeled with $I$
**create** new automaton $\mathcal{A}_D = (\{q\}, \emptyset, \{q\}, \emptyset)$    //$Q_D, \delta_D, I_D, F_D$
**add** $q$ to an initially empty set $C$   //$C$ is "to-construct"
**while** $C \neq \emptyset$ **do**
  **remove** a state $r$ (with its label $T$), from $C$
  **if** $T \cap F \neq \emptyset$ **then**
    $F_D \leftarrow F_D \cup \{r\}$
  **create** set of extended symbols $E = \emptyset$
  **for** each edge $(s, \sigma, d) \in \delta \mid s \in T$ **do**
    **add** $\sigma$ to $E$
  **for** each extended symbol $\sigma_i$ in $E$ **do**
    **create** a new label $U = \emptyset$
    **for** each edge $(s, \sigma, d) \in \delta \mid s \in T, \sigma = \sigma_i$ **do**
      $U \leftarrow U \cup \{d\}$
    **if** $U \neq \emptyset$ **then**
      **find** state $u \in Q_D$ with label $U$
      **if** $\nexists u$ **then**
        **create** new state $u \in Q_D$ labeled with $U$
        $C \leftarrow C \cup \{u\}$
      //
      $\delta_D \leftarrow \delta_D \cup \{(r, \sigma_i, u)\}$
**return** $\mathcal{A}_D$   //$|I_D|$=1, as required for Intersect()
```

**Algorithm 4.1:** Weak determinization algorithm. Differences with the strong determinization in Algorithm 4.2 are highlighted in gray.

other symbols. During these manipulations, the dual-layer symbol alphabet helps to perform Boolean simplifications not seen by a single layer alphabet.

Algorithms 4.1 and 4.2 present the algorithms for creating weakly and strongly deterministic automata respectively, using the symbolic alphabet. Both algorithms are based on the subset construction technique, and although they may appear similar, the main difference between the two concerns the depth of symbols they manipulate. The algorithm for weak determinization only analyses the extended symbols, whereas the strong determinization pushes deeper into the primary symbols. A state in the determinized automaton is labeled by a totally ordered subset of states from the input automaton. In the implementation, the total order relation is that which is produced by sorting the actual data elements that represent states (state numbers).

Both algorithms start by applying the total order to the set of initial states in

```
 1: FUNCTION: StrongDeterminize($\mathcal{A}$)
 2: //a label is a totally ordered set $T \subseteq Q$
 3: **apply** total order to $I$
 4: **create** new state $q$ labeled with $I$
 5: **create** new automaton $\mathcal{A}_D = (\{q\}, \emptyset, \{q\}, \emptyset)$    $//Q_D, \delta_D, I_D, F_D$
 6: **add** $q$ to an initially empty set $C$   $//C$ is "to-construct"
 7: **while** $C \neq \emptyset$ **do**
 8:    **remove** a state $r$ (with its label $T$), from $C$
 9:    **if** $T \cap F \neq \emptyset$ **then**
10:       $F_D \leftarrow F_D \cup \{r\}$
11:    **create** set of primary symbols $P = \emptyset$
12:    **for** each edge $(s, \sigma, d) \in \delta \mid s \in T$ **do**
13:       **add** $\sigma$'s primary symbol(s) to $P$
14:    **for** each assignment $\omega$ of primary symbols in $P$ **do**
15:       **create** a new label $U = \emptyset$
16:       **for** each edge $(s, \sigma, d) \in \delta \mid s \in T, (\sigma)_\omega = \texttt{true}$ **do**
17:          $U \leftarrow U \cup \{d\}$
18:       **if** $U \neq \emptyset$ **then**
19:          **find** state $u \in Q_D$ with label $U$
20:          **if** $\nexists u$ **then**
21:             **create** new state $u \in Q_D$ labeled with $U$
22:             $C \leftarrow C \cup \{u\}$
23:          $\sigma_i \leftarrow$ **create** or **retrieve** symbol in $\Sigma$ for $\omega$
24:          $\delta_D \leftarrow \delta_D \cup \{(r, \sigma_i, u)\}$
25: **return** $\mathcal{A}_D$
```

**Algorithm 4.2:** Strong determinization algorithm. Differences with the weak determinization in Algorithm 4.1 are highlighted in gray.

their argument automaton (line 3). The first state $q$ in the determinized automaton is created, and labeled with this ordered set of initial states (line 4). The state actually comprises the result automaton's initial topology, where it is also added to the set of initial states (line 5). This state is then used to initiate the subset construction loop in line 7, whereby the central element is the state creation set $C$. The subset construction implicitly creates an automaton with a single initial state, a feature that is required by the intersection algorithm's use of weak determinization.

Inside the state creation loops in both algorithms (lines 8 to 24), new states are created to represent subsets of states in the input NFA to be determinized (hence the name *subset construction*). When any state in a subset is a final state in the input automaton, the state corresponding to the subset is also a final state (lines 9 and 10).

The differences between strong and weak determinization start in line 11 in both algorithms. In weak determinization, lines 11 to 13 build the set of extended symbols that are used in the outgoing edges of the subset of states. The subset being referred to is from the input automaton, and is being used to construct the new state in the determinized automaton. In strong determinization, lines 11 to 13 instead build the set of *primary* symbols used in the outgoing edges of the subset of states. In both algorithms, if the set $P$ is the empty set, then no edges are created for the state and the **for** loop in line 14 does not execute its block of statements.

In the strong (weak) algorithm, for each unique assignment of primary symbols (for each extended symbol), the set of destination states is built in lines 15 to 17; this becomes the label for the destination of a new edge that is to be created for the given assignment of primary symbols (extended symbol). If this destination state already exists, it is used, else it is created (lines 19 to 21) and is added to the construction set $C$ in line 22. The new edge is created in lines 23 and 24, and the entire process is repeated until there are no more states left to create in $C$. The algorithms are proven to terminate since only a finite number of states (although exponential) can be added to the construction set. Determinization produces an automaton with $O(2^n)$ states in the worst case, where $n$ is the number of states in the input automaton. In practice, the resulting automaton is often reasonable in size.

### 4.2.3 Minimization of Automata

Procedures exist for minimizing conventional automata, such as Hopcroft's $n \log n$ algorithm [100], and Brzozowski's

$$\text{Reverse} \rightarrow \text{Determinize} \rightarrow \text{Reverse} \rightarrow \text{Determinize} \tag{4.1}$$

algorithm [37, 177], called the RDRD algorithm herein. The Hopcroft algorithm applies to complete DFAs, where the transition relation $\delta$ is completely specified for every symbol at every state, whereas the RDRD algorithm can accept an incomplete NFA. In both cases, a minimized DFA is produced. As a side note, the RDRD algorithm does not modify the language accepted by an automaton given that the reversal is performed twice, and that determinization itself does not change the language accepted.

The Reverse() algorithm used in minimization exchanges $s$ with $d$ for all edges $(s, \sigma, d) \in \delta$, and also exchanges the initial and final state sets, as shown in the algorithm in Algorithm 4.3. Since the automaton definition (Definition 4.3) allows the set of final states to be empty, which occurs in the automaton for the null sequence

```
1: FUNCTION: REVERSE($\mathcal{A}$)   //$(Q, \delta, I, F)$
2: if $F = \emptyset$ then
3:     create new automaton $\mathcal{A}_1 = (\{q_1\}, \emptyset, \{q_1\}, \emptyset)$
4: else
5:     create new automaton $\mathcal{A}_1 = (Q, \delta, F, I)$
6:     for each edge $(s, \sigma, d) \in \delta_1$ do
7:         $(s, \sigma, d) \leftarrow (d, \sigma, s)$
8: return $\mathcal{A}_1$
```

**Algorithm 4.3:** Automaton reversal algorithm.

for example, the algorithm treats this case separately in line 3. A subtlety also exists in line 5 in the reversal algorithm, where sets $I$ and $F$ are swapped.

In this subsection, the minimization of automata based on the symbolic alphabet is developed. Although conventional DFAs can be provably minimized, no claim is made about the minimalism of the automata used in this work, which can be nondeterministic in some cases. Producing minimized NFAs [115], let alone NFAs with Boolean expressions encoded in the symbols, is a hard problem. The approach used here is more a heuristic for reducing the size of automata with symbolic alphabets.

The minimization procedure has the effect of pruning any unconnected or useless connected states which can result from the application of other algorithms. The minimization of symbolic automata is based on an observation that even though conventional minimization only applies to DFAs, in many cases the required determinization (worst case exponential) is not a big penalty. In most cases the increase in states is quite modest, and in some cases determinization even produces smaller automata; this behavior is also observed in the weighted automata used in speech recognition [39].

Algorithm 4.4 presents the algorithm used to minimize symbolic automata. The minimization approach is based on the RDRD algorithm from (4.1), which corresponds to lines 2, 3, 5 and 7 in Algorithm 4.4. Weak determinization is employed for the determinization step, as opposed to strong determinization. The guiding principle in minimizing automata is that any form of nondeterminism is tolerated, and even sought when it causes the resulting automaton to be smaller. Because of this principle, the final determinization in the RDRD strategy is only applied if it reduces the number of states (lines 9 and 10).

Circuits are well suited for implementing nondeterministic conventional RE pattern matchers [163], and in this respect, the automata used here with Booleans as the alphabet are no different. Surprisingly, the weak determinization used implies that

```
1: FUNCTION: MINIMIZE($\mathcal{A}$)
2: $\mathcal{A} \leftarrow$ REVERSE($\mathcal{A}$)
3: $\mathcal{A} \leftarrow$ WEAKDETERMINIZE($\mathcal{A}$)
4: COLLAPSEFINALSTATES($\mathcal{A}$)
5: $\mathcal{A} \leftarrow$ REVERSE($\mathcal{A}$)
6: COLLAPSEFINALSTATES($\mathcal{A}$)
7: $\mathcal{A}_2 \leftarrow$ WEAKDETERMINIZE($\mathcal{A}$)
8: COLLAPSEFINALSTATES($\mathcal{A}_2$)
9: if $|Q_2| < |Q|$ then
10:     $\mathcal{A} \leftarrow \mathcal{A}_2$
11: return $\mathcal{A}$
```

**Algorithm 4.4:** Automaton minimization algorithm.

the minimization algorithm, when applied to an NFA, can produce an automaton that in the worst case has an exponential number of states.

As will be seen in Chapter 5, when a sequence is used as a property, a strongly deterministic automaton is produced for that sequence. When the minimization algorithm is applied to a property's automaton that contains a strongly deterministic sub-portion, the weak determinization used in minimization has no effect on that portion of the automaton; however, when other parts of a property automaton that have not undergone strong determinization are minimized, the weak determinization preserves a certain amount of nondeterminism, and helps keep the checkers more compact.

Another key factor in the minimization algorithm is the collapsing of final states in presence of self-loop edges with the true symbol (lines 4, 6 and 8 in Algorithm 4.4). The true edge differentiates itself from other edges because it is guaranteed to cause a transition at each cycle in the automaton. Incorporating knowledge of this fact in the minimization algorithm helps to implement another type of optimization, namely the collapsing of final states. The reason there are two calls to this function in the second half of the RDRD strategy is that depending on the outcome of the comparison, the algorithm may if fact implement RDR, and in this case the final reversal should also have a collapsing of final states. These are performed separately after the last reversal and after the last determinization in order for the automaton size comparison to be the most meaningful.

The COLLAPSEFINALSTATES() algorithm used in minimization is presented in Algorithm 4.5. This algorithm removes a subset of states that is closed under the true self-loop. The first part of the algorithm (lines 2 to 5) works by detecting a true transition $(s, \mathsf{true}, d)$ between two final states, where state $d$ has a true self-loop. The

```
1: FUNCTION: CollapseFinalStates($\mathcal{A}$)   //$(Q, \delta, I, F)$
2: while $\exists$ pair of edges $(s_1, \sigma_1, d_1)$ and $(s_2, \sigma_2, d_2)$ | $\{s_1, s_2, d_1, d_2\} \subseteq F$, $\sigma_1 = \sigma_2 =$
   true, $d_1 = s_2 = d_2$ do
3:    for each edge $(s_3, \sigma_3, d_3)$ | $s_3 = s_1$ do
4:       $\delta \leftarrow \delta - \{(s_3, \sigma_3, d_3)\}$
5:    $\delta \leftarrow \delta \cup \{(s_1, \sigma_1, s_1)\}$
6: while $\exists$ pair of edges $(s_1, \sigma_1, d_1)$ and $(s_2, \sigma_2, d_2)$ | $\{s_2, d_1, d_2\} \subseteq F$, $s_1 \nsubseteq F$,
   $\sigma_1 = \sigma_2 =$true, $d_1 = s_2 = d_2$ do
7:    for each edge $(s_3, \sigma_3, d_3)$ | $s_3 = s_1$, $\sigma_3 \neq \sigma_1$, $d_3 \neq d_1$ do
8:       $\delta \leftarrow \delta - \{(s_3, \sigma_3, d_3)\}$
9: return $\mathcal{A}$
```

**Algorithm 4.5:** Algorithm for collapsing final states.

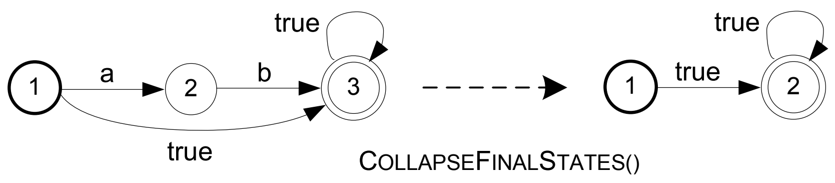


**Figure 4.5:** Collapsing final states with true edges.

true self-loop can be safely replicated on the source state because it adds nothing to the language accepted by the automaton (line 5). Any other outgoing edges from this source state can be safely removed in lines 3 and 4 because once a final state becomes active, the true self-loop will perpetually keep it active, and there is no point in triggering other states.

The second part of the algorithm in lines 6 to 8 performs another type of optimization with regards to the true edge. It is very similar to the first part, with the exception that the first state in the state pair is not a final state. The second optimization works by detecting a true transition $(s, \text{true}, d)$ between two states, where state $s$ is non-final and state $d$ is a final state with a true self-loop. In such cases, any other outgoing transitions from state $s$ are pointless because the true transition will activate a perpetually active final state in the next transition, thus the other outgoing transitions can be removed (lines 7 and 8). If any state becomes unconnected during the collapsing of final states, it is removed from the automaton (not shown in the algorithm). The second type of optimization is illustrated in Figure 4.5, where the $a$ edge can be removed given the simultaneous true transition to a perpetually active final state. The first type of optimization is illustrated in a complete example.

**Example 4.4:** The following assertion is used to show the effect of collapsing final

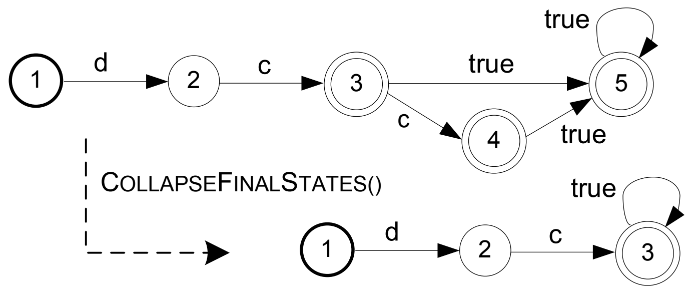


**Figure 4.6:** Example for collapsing final states.

states in minimization.

assert never $\{c$[*1:2];$d\}$;

This example is illustrated in Figure 4.6 and Figure 4.7. The actual steps used to create the automaton require the notions in Chapter 5, and are not topical at this point. The top part of Figure 4.6 shows the automaton as it appears after the first determinization in the minimization algorithm (hence it was also reversed). The true self-loop on state 5, and the true edge from state 3 to 5, combined with the fact that states 3 and 5 are final states implies that once state 3 is reached, the automaton will remain active for the remainder of execution, or until the checker is reset. Since this automaton has undergone a reversal, this is expected in the example; i.e. the loop created by the never operator is now at the end of the automaton. As a result, the transitions to and from state 4 are in effect useless. Consequently, the smaller automaton at the bottom of Figure 4.6 is produced when the collapsing of final states is performed. That automaton shows the state of the computation after line 4 is executed in the minimization algorithm.

To show the global effect of collapsing final states in presence of true edges, if it is not performed when minimizing the automaton for the assertion above, the four-state automaton in Figure 4.7 a) is generated. With the optimization, the smaller three-state automaton in Figure 4.7 b) is produced. This reduction actually reveals more insight into the run-time semantics of PSL for dynamic verification. If the assertion did not have the never operator, the sequence would only be checked starting in the first cycle of execution, and the full language defined by the sequence $\{c$[*1:2];$d\}$ would apply. However, in presence of the never operator, this optimization shows that only the $\{c;d\}$ portion needs to be detected. In other words, when it is continually triggered, $\{c;d\}$ temporally subsumes the larger $\{c;c;d\}$.

The minimization algorithm will not be called explicitly in the automaton imple-

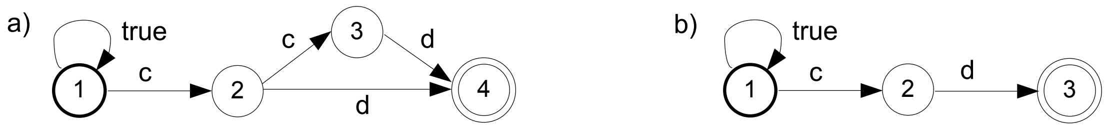


**Figure 4.7:** Effect of collapsing final states in minimization, for example 4.4. a) Without CollapseFinalStates(), b) with CollapseFinalStates().

mentation of properties and sequences in the next chapter. In the checker generator, minimization is applied to top-level sequences, as opposed to recursively at each level in a sequence's operators, with one exception. Intersection involves a product automaton construction and it is beneficial to minimize the argument automata beforehand. As will be seen in the implementation of sequences, all other sequence operators have a linear space complexity, and minimization can be safely relegated to its highest-level point in the sequence's expression. Minimization is also applied at the property level, but for simplicity will also not explicitly appear in the implementation of properties. Functionally, minimization could in fact only be called once before the automaton is actually converted to RTL code to form the checker.

### 4.2.4 Complementation of Automata

Complementing (or negating) an automaton corresponds to creating a new automaton that matches the complement language. In the case of symbolic automata over Booleans, the complement language corresponds to all the traces that are not accepted by the original automaton.

Algorithm 4.6 presents the algorithm used to complement automata with the symbolic alphabet. This algorithm is also used in an example in the next chapter (Example 5.1). The strong determinization with completion is performed by a separate algorithm that is explained thereafter. Although the automata complementation algorithm is not used in the checker generator, it is presented nonetheless so that the automaton framework over Booleans is fully defined for the conventional operators.

The Negate() algorithm implements the same type of procedure as described in Section 2.3 for complementing conventional automata. The two steps for determinization and completion are implemented in one function in line 2, and the complementation of final states is performed in line 3.

A strong determinization algorithm that produces complete DFAs is presented in Algorithm 4.7. Although lines 28 and 31 can be factored from the **if/else** in lines 23 and 30, they are kept redundant in order for the algorithm to appear as similar

```
1: FUNCTION: NEGATE($\mathcal{A}$)
2: $\mathcal{A} \leftarrow$ STRONGDETERMINIZEWITHCOMPLETION($\mathcal{A}$)   $//Q, \delta, I, F$
3: $F \leftarrow Q - F$
4: return $\mathcal{A}$
```

**Algorithm 4.6:** Automata negation algorithm.

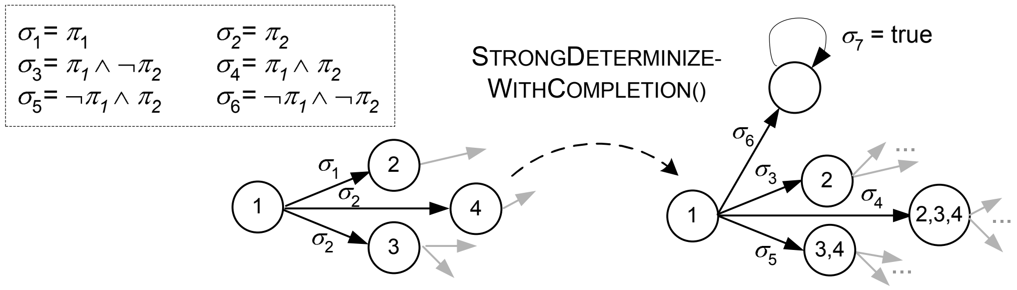

$\sigma_1 = \pi_1$
$\sigma_2 = \pi_2$
$\sigma_3 = \pi_1 \wedge \neg\pi_2$
$\sigma_4 = \pi_1 \wedge \pi_2$
$\sigma_5 = \neg\pi_1 \wedge \pi_2$
$\sigma_6 = \neg\pi_1 \wedge \neg\pi_2$
STRONGDETERMINIZE-WITHCOMPLETION()
$\sigma_7$ = true
1
2
3
4
2,3,4
3,4
$\sigma_1$
$\sigma_2$
$\sigma_3$
$\sigma_4$
$\sigma_5$
$\sigma_6$
...

**Figure 4.8:** Strong determinization with completion.

as possible to the strong determinization algorithm in Algorithm 4.2; only 6 lines were added (shown in light gray) to that algorithm to yield a version which produces complete DFAs for the symbolic alphabet. The algorithm adds a dead state $f$ whereby all Boolean conditions not taken by outgoing edges in a state are routed to this dead state using supplemental edges (lines 31 and 32). The dead state $f$ does not require a label because it does not take part in the subset construction, and is directly added to the resultant automaton (lines 5 and 7). In order for the dead state to also adhere to the requirements of completeness, a self-loop with a true edge is added in line 7.

The key to proving that the algorithm produces complete automata is that for every state that is created in the new automaton in the subset construction, the **for** loop in line 19 combined with the **if/else** in lines 23 and 30 add the necessary edges for covering all the Boolean possibilities for outgoing transitions in a given state. The only state that is not part of the subset construction is also complete given the self-loop with true that is added at the beginning of the algorithm.

Normally the complete-determinization algorithm is not used because the dead state $f$ would get pruned during minimization. It is therefore important that minimization must not be performed between lines 2 and 3 in the NEGATE() algorithm, in order for the complete DFA to remain intact.

Figure 4.8 shows how the complete-determinization algorithm operates on a subportion of an automaton. The automaton on the left is identical to the one in Figure 4.4 used to illustrate the differences between strong and weak determinization; it is used here to show how a complete strongly deterministic automaton can be

```
FUNCTION: StrongDeterminizeWithCompletion($\mathcal{A}$)
//a label is a totally ordered set $T \subseteq Q$
**apply** total order to $I$
**create** new state $q$ labeled with $I$    $//Q_D, \delta_D, I_D, F_D$
**create** new state $f$
$\sigma_t \leftarrow$ **create** or **retrieve** symbol in $\Sigma$ for true
**create** new automaton $\mathcal{A}_D = (\{q, f\}, \{(f, \sigma_t, f)\}, \{q\}, \emptyset)$
**add** $q$ to an initially empty set $C$   $//C$ is "to-construct"
**while** $C \neq \emptyset$ **do**
   **remove** a state $r$ (with its label $T$), from $C$
   **if** $T \cap F \neq \emptyset$ **then**
      $F_D \leftarrow F_D \cup \{r\}$
   **create** set of primary symbols $P = \emptyset$
   **for** each edge $(s, \sigma, d) \in \delta \mid s \in T$ **do**
      **add** $\sigma$'s primary symbol(s) to $P$
   **if** $P = \emptyset$ **then**
      $\delta_D \leftarrow \delta_D \cup \{(r, \sigma_t, f)\}$
   **else**
      **for** each assignment $\omega$ of primary symbols in $P$ **do**
         **create** a new label $U = \emptyset$
         **for** each edge $(s, \sigma, d) \in \delta \mid s \in T$, $(\sigma)_\omega =$ true **do**
            $U \leftarrow U \cup \{d\}$
         **if** $U \neq \emptyset$ **then**
            **find** state $u \in Q_D$ with label $U$
            **if** $\nexists u$ **then**
               **create** new state $u \in Q_D$ labeled with $U$
               $C \leftarrow C \cup \{u\}$
            $\sigma_i \leftarrow$ **create** or **retrieve** symbol in $\Sigma$ for $\omega$
            $\delta_D \leftarrow \delta_D \cup \{(r, \sigma_i, u)\}$
         **else**
            $\sigma_i \leftarrow$ **create** or **retrieve** symbol in $\Sigma$ for $\omega$
            $\delta_D \leftarrow \delta_D \cup \{(r, \sigma_i, f)\}$
**return** $\mathcal{A}_D$
```

**Algorithm 4.7:** Algorithm for strong determinization with completion. Additions compared to the strong determinization algorithm in Algorithm 4.2 are highlighted in gray.

produced. As before, only one state is explored in full detail and the remainder of the automaton is not shown. The complete strongly deterministic version is similar to the strongly deterministic one, with the exception that an extra outgoing edge is added to state 1. This edge leads to the dead state, which is not labeled because it does not represent a subset of states of the input automaton. For the two states that are completely defined in the automaton in the right side of the figure, namely the unlabeled state and state 1, it can be verified that *a single* outgoing transition is taken in each state for any given status of the Boolean primitives; this is consistent with the definition of complete DFAs (Definition 2.5).

## 4.3 Generating Circuit-Level Checkers from Assertion Automata

Even though the algorithms for expressing PSL statements in automaton form are treated in the next chapter, the automaton definition and examples contained in this chapter are sufficient prerequisites for introducing the transformation of automata into circuit-level checkers. The main result that must be developed is a procedure for transforming the automaton into a form suitable for in-circuit assertion monitoring. Furthermore, the automata may not be entirely (if at all) strongly deterministic. Circuits composed of combinational and sequential logic represent an ideal implementation because each state's computations can be performed in parallel. In contrast, traditional software implementations make heavy use of deterministic automata so that the software can easily track a single active state.

The procedure described in this section defines the operator $\mathcal{H}(\mathcal{A})$, which represents the creation of HDL code from a given argument, which in this case is an automaton. If $b$ is a PSL Boolean, the notation $\mathcal{H}(b)$ is also valid and will be used in the automaton implementation of Booleans in the next chapter. The resulting circuit descriptions from $\mathcal{H}()$ become the checkers that are responsible for monitoring the behavior expressed by the assertions.

Implementing an automaton in RTL is performed using the following encoding strategy for each state:

1. A *state signal* is defined as a Boolean disjunction of the *edge signals* that hit a given state;
2. An *edge signal* is a Boolean conjunction of the expression represented by the edge's symbol with the *sampled state signal* from which the edge originates;

3. A *state signal* is sampled by a flip-flop, and the flip-flop's output is referred to as the *sampled state signal*;
4. If a state is an initial state, the flip-flop defined in step 3 is initially set, else it is initially reset;
5. The *assertion signal* (or checker output) is a Boolean disjunction of the *state signals* of all the final states.

In sum, automata are implemented in hardware using combinational logic and flip-flops. Since an entire automaton is rarely ever strongly deterministic, any subset of its $n$ states can simultaneously be active, thus one flip-flop per state (i.e. $n$ flip-flops) is the most efficient encoding. Even if a portion of an automaton is strongly deterministic, multiple activations from other portions could retrigger it (and overlap within it) thus allowing more than one active state, and consequently, disallowing the $\lceil \log_2 n \rceil$ flip-flop encoding. It is true however, that if an entire assertion automaton with $n$ states is strongly deterministic, then at most $\lceil \log_2 n \rceil$ flip-flops could be used to keep track of *the* active state; however, this comes at the expense of more combinational logic since the state vector needs to be decoded for use in the edge signals.

The flip-flops in the automaton states have the additional property that they can be reset by the global reset signal. This way, when the circuit under verification is reset, the checkers can also be reset to provide the correct monitoring. Expressing the RTL circuit as HDL code in the desired language, or even in schematic form, follows naturally. The following example shows how automata are converted to circuit form to produce circuit-level assertion checkers.

**Example 4.5:** The following bus arbiter assertion states that when the request ($req$) signal goes high, the arbiter should give a bus grant ($gnt$) within at most three cycles, and $req$ must be kept high until (but not including) the cycle where the grant is received.

$$\textsf{assert always } \{\sim req\ ;\ req\}\ |\!=\!>\ \{req[*0{:}2]\ ;\ gnt\};$$

The example assertion is compiled into the automaton shown at the top of Figure 4.9. The automaton is then expressed in RTL as the circuit shown in the bottom of the figure. The labels containing Boolean expressions can also be expressed using logic circuits, but are kept in text form for simplicity in this example. The always operator has the effect of creating the self loop on state 1, which continually activates the automaton in search of a faulty trace. This state is optimized away given that it is an initial state and that the true self loop perpetually keeps it active. As was shown in

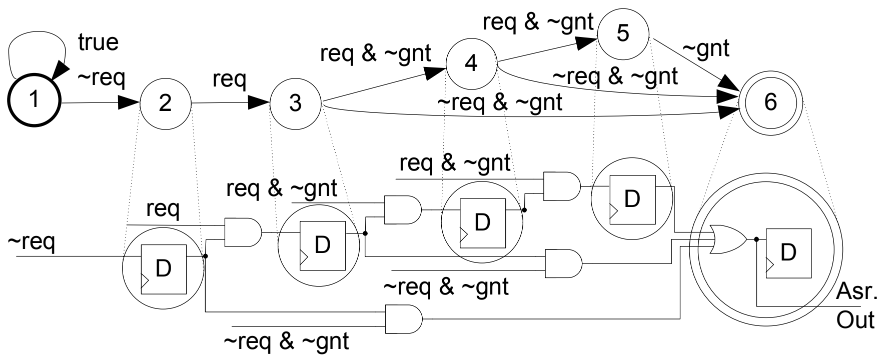


**Figure 4.9:** Circuit-level checker for assertion in Example 4.5.

Figure 2.2, such RTL checkers can then be connected to the circuit-under verification to detect functional errors.

By default a flip-flop is added to the assertion signal to improve the frequency performance of the checker (not shown in the example). This is particularly useful when a checker has many final states or when final states have many incoming edges. The flip-flop is also beneficial when the assertion signal is routed to a pin or a register file elsewhere on the circuit being tested. The checker should interfere as little as possible with the source design's timing, and the flip-flop helps reduce the chance that the assertion signals are on the critical path. The penalty is a one cycle delay in the response of the checker. The FF can be avoided by using the appropriate switch in the checker generator, when desired.

# Chapter 5

# Automata Implementation of Assertions

## 5.1 Introduction and Overview

The automaton framework for checkers was outlined in the previous chapter, where determinization and minimization issues were the two most significant issues treated. The current chapter presents the construction of automata for representing PSL expressions. The four main constituents of the PSL language, namely Booleans, sequences, properties and verification directives are each handled in separate sections. Boolean expressions, sequences and properties alone accomplish nothing in most verification tools, and must be used in verification directives. The bottom-up construction therefore begins with PSL Booleans and ends with verification directives. The underlying expectation is to produce the smallest automata so that when converted to RTL circuits, the checkers utilize fewer hardware resources.

Automata for Booleans are constructed by first converting their expressions to HDL code, and then building a simple two-state automaton with the Boolean's symbol on the edge. Automata for Booleans are the building blocks of sequences, both of which are building blocks of properties. Automata for sequences and properties are constructed directly using a variety of algorithms, or indirectly using rewrite rules. Rewrite rules are a syntactic transformation and are used to avoid the need for explicitly supporting an operator. When no rewrite rules apply, operators are handled with specific algorithms.

To provide more useful debug information, the automata algorithms are designed such that the assertion signal does not simply indicate a yes/no answer obtained at the end of execution, but rather a continuous and dynamic report of when the assertion has failed throughout the execution run (as discussed in Section 2.2). Although this chapter focuses on automata construction for PSL statements, building the actual checkers requires only the subsequent application of the automaton-to-circuit conversion developed in Section 4.3.

## 5.2 Implementation of Booleans

The building blocks of an assertion are Boolean expressions over the signals in the circuit under verification. Whether in sequences or properties, Booleans are the lowest constituents in the PSL language hierarchy. The goal in this section is to implement all the Booleans items in Definition 2.7 as HDL code, and subsequently to implement automata for Booleans.

In the following, $\mathcal{H}$ represents a portion of HDL code, and $\mathcal{H}(x)$ represents the conversion of an argument $x$ into HDL code, and thus also represents a portion of HDL code. The argument can be a PSL expression or an automaton, depending on the case. The conversion of an automaton to HDL code, denoted $\mathcal{H}(\mathcal{A})$, was presented in Section 4.3. The construction of an automaton from a never property is used below in the implementation of ended(), but will be presented further in Section 5.4. The default clock declaration is implemented in Section 5.5, and for now it is assumed that a *global clock string* holds the name of the clock signal in effect.

To recall, the symbol $\mathcal{A}$ is used to represent an arbitrary automaton, as defined in Definition 4.3, and the functional form $\mathcal{A}(x)$ represents the construction of an automaton from an argument expression $x$, and is also an automaton because the return value is an automaton. A superscript, such as in $\mathcal{A}^B(x)$, represents additional information to indicate what type of construction is desired (in this case Boolean). The other superscripts used are $^S$, $^P$ and $^V$ for sequences, properties, and verification directives respectively. The resulting automata with superscripts also conform to the definition of automata given in the previous chapter.

Implementing the Boolean layer is done in two parts: first the conversion of a Boolean to HDL is presented, and afterwards the expression of an automaton for a Boolean, as used in a sequence or a property, is derived.

The conversion of Booleans to HDL is presented below in Proposition 5.1, in a manner that resembles the syntax from Definition 2.7 as closely as possible, so that the link between syntax and implementation can be better observed. In the following, non-italicized prefixes separated by an underscore represent an additional type constraint, and do not represent a new type as such.

**Proposition 5.1:** If $b$ is a Boolean, $i$ is a nonnegative integer, $e$ is an Expression, $f$ is a built-in function and $s$ is a Sequence then the HDL conversion of Booleans defined in Definition 2.7 is denoted $\mathcal{H}(b)$, and is performed as follows:

$\mathcal{H}(b)$ :

- $\mathcal{H}(\text{boolean\_}e) \quad = \quad$ [*defined in next line*]

$\mathcal{H}(e)$ :

- $\mathcal{H}(\text{Verilog\_}e) \quad = \quad e$
- $\mathcal{H}(b_1 \texttt{ -> } b_2) \quad = \quad (\sim b_1 \mid b_2)$
- $\mathcal{H}(b_1 \texttt{ <-> } b_2) \quad = \quad (\,(\sim b_1 \mid b_2) \mathbin{\&} (\sim b_2 \mid b_1)\,)$
- $\mathcal{H}(\mathsf{true}) \quad = \quad \mathsf{1'b1}$
- $\mathcal{H}(\mathsf{false}) \quad = \quad \mathsf{1'b0}$
- $\mathcal{H}(f) \quad = \quad$ [*defined in next line*]

$\mathcal{H}(f)$ :

- $\mathcal{H}(\mathsf{prev}(e)) \quad = \quad \textsc{Dff}(e)$
- $\mathcal{H}(\mathsf{prev}(e,\, i)) \quad = \quad \textsc{Dff}^{i}(e)$
- $\mathcal{H}(\mathsf{rose}(\text{bit\_}e)) \quad = \quad (\sim\textsc{Dff}(e) \mathbin{\&} e)$
- $\mathcal{H}(\mathsf{fell}(\text{bit\_}e)) \quad = \quad (\textsc{Dff}(e) \mathbin{\&} \sim e)$
- $\mathcal{H}(\mathsf{onehot}(\text{bit\_vector\_}e)) \quad = \quad (\,((e{-}1) \mathbin{\&} e) == 0)$
- $\mathcal{H}(\mathsf{onehot0}(\text{bit\_vector\_}e)) \quad = \quad (\,\mathcal{H}(\mathsf{onehot}(e)) \mathbin{||} (e == 0)\,)$
- $\mathcal{H}(\mathsf{stable}(e)) \quad = \quad (\textsc{Dff}(e) == e)$
- $\mathcal{H}(\mathsf{ended}(s)) \quad = \quad \mathcal{H}(\mathcal{A}^{P}(\mathsf{never}\ s))$

The items that have italicized comments in brackets do not actually need to be implemented and are only left to show the correspondence to Definition 2.7. Each item from Proposition 5.1 will be presented and explained below, immediately following its appearance.

$$\mathcal{H}(\text{Verilog\_}e) \quad = \quad e$$

In the case where a Verilog expression is used as a Boolean, it is simply returned directly given that the Boolean layer is built upon the underlying HDL, which corresponds to Verilog HDL code in this case.

$$\begin{aligned} \mathcal{H}(b_1 \texttt{ -> } b_2) \quad &= \quad (\sim b_1 \mid b_2) \qquad (5.1) \\ \mathcal{H}(b_1 \texttt{ <-> } b_2) \quad &= \quad (\,(\sim b_1 \mid b_2) \mathbin{\&} (\sim b_2 \mid b_1)\,) \end{aligned}$$

Implication and equivalence are not part of the Verilog language as such, and are added to the definition of Booleans. These operators are expressed in Verilog using their respective Boolean equivalencies, as shown on the right above.

$$\begin{aligned}\mathcal{H}(\mathsf{true}) &= \mathsf{1'b1}\\ \mathcal{H}(\mathsf{false}) &= \mathsf{1'b0}\end{aligned}$$

Symbols true and false are expressed in HDL as single digit binary constants. The remaining cases from Proposition 5.1 are the built-in functions, described next.

$$\begin{aligned}\mathcal{H}(\mathsf{prev}(e)) &= \textsc{Dff}(e)\\ \mathcal{H}(\mathsf{prev}(e,\, i)) &= \textsc{Dff}^i(e)\end{aligned}$$

In any given clock cycle $n$, the prev operator evaluates to the value of the argument expression at cycle $n-i$. The implementation for this operator is denoted symbolically above using the $\textsc{Dff}^i(e)$ function, and in reality corresponds to the instantiation of a register chain (or pipeline) with $i$ stages in depth. The default clock is implicitly used as the clock signal for any flip-flop represented by the $\textsc{Dff}()$ notation; the input of the flip-flop is the argument and the return value of $\textsc{Dff}()$ represents the output of the flip-flop. Thus, the resulting expression of $\textsc{Dff}^i()$ evaluates to the signal or vector of the last register(s) in the chain. When the exponent is not specified, $i = 1$ is assumed. The register pipeline has the same width as the argument expression $e$. This register chain is created as actual HDL code using Verilog non-blocking assignments and the clock signal that exists in the global clock string at that time. For the case $i = 0$, no registers are created and the expression is returned directly.

It should be noted that the flip-flops generated by $\textsc{Dff}^i(e)$ are not reset with the checker's reset, and even if a builtin function is used under the scope of an abort operator, the abort has no effect on the flip-flops. In other words, these flip-flops are never reset, and are used purely to create a clock cycle delay. It is up to the implementation to determine their initial state. A flip-flop is in an undetermined state until a proper input value can be latched by the flip-flop.

$$\begin{aligned}\mathcal{H}(\mathsf{rose}(\mathrm{bit\_}e)) &= (\sim\textsc{Dff}(e)\ \&\ e)\\ \mathcal{H}(\mathsf{fell}(\mathrm{bit\_}e)) &= (\textsc{Dff}(e)\ \&\ {\sim}e)\\ \mathcal{H}(\mathsf{stable}(e)) &= (\textsc{Dff}(e) == e)\end{aligned}$$

The above functions relate the value of an expression to the value in its previous clock cycle, and also use the HDL flip-flop declaration mentioned above. The rose() and fell() functions operate on single-bit expressions only, whereas bit vectors can be used in stable().

$$\begin{aligned}\mathcal{H}(\mathsf{onehot}(\text{bit\_vector\_}e)) &= (\ ((e{-}1)\ \&\ e) == 0) \\ \mathcal{H}(\mathsf{onehot0}(\text{bit\_vector\_}e)) &= (\ \mathcal{H}(\mathsf{onehot}(e))\ ||\ (e == 0)\ )\end{aligned}$$

The implementation of the onehot function takes advantage of the arithmetic operators that can be specified in languages such as Verilog. Detecting the cases where a single bit is active in a sequence of bits is performed with the “bit trick” involving decrementation and word-level intersection. The onehot0 function also allows for all-zeros to occur, and is implemented using onehot() with an additional Boolean expression.

$$\mathcal{H}(\mathsf{ended}(s)) \quad = \quad \mathcal{H}(\mathcal{A}^P(\mathsf{never}\ s))$$

The ended function evaluates to logic-1 every time the argument sequence is matched. This is modeled using the never property, which triggers whenever its sequence argument is detected. The implementation of properties is treated further in this chapter, and must unavoidably be assumed at this point.

The items contained in Proposition 5.1 show how any PSL Boolean can be expressed at the HDL level. Constructing an automaton for the matching of a Boolean, as required in sequences and properties, is described next in Proposition 5.2.

**Proposition 5.2:** If $b$ is a Boolean and $\mathcal{H}(b)$ represents the HDL expression for $b$, as presented in Proposition 5.1, then the construction of an automaton for Boolean $b$, denoted $\mathcal{A}^B(b)$, is performed as follows:
$\mathcal{A}^B(b)$ :

- $\mathcal{A}^B(b) \quad = \quad \textsc{BaseCase}(\mathcal{H}(b))$

An implicit consequence of the separate implementations in Proposition 5.1 and Proposition 5.2 is that automata are built only for top-level Booleans as opposed to building automata for arbitrary sub-expressions of a complex Boolean expression.

The algorithm for creating an automaton for a Boolean's HDL expression is described in Algorithm 5.1. This is the base case for the inductive automaton construction procedure for sequences. The algorithm takes an HDL expression as argument, and proceeds to build an extended symbol to represent it. Top-level negations are removed in line 2 and added in line 5 when necessary. The primary symbol table is used in line 3, whereas the extended symbol table is used in lines 5 and 7. This is

```
1: FUNCTION: BaseCase($\mathcal{H}$)
2: $\mathcal{H} \leftarrow$ **remove** outermost negations from $\mathcal{H}$
3: $\pi \leftarrow$ **create** or **retrieve existing** symbol in $\Pi$ for $\mathcal{H}$
4: **if** odd number of outermost negations were removed in line 2 **then**
5:     $\sigma \leftarrow$ **create** or **retrieve existing** symbol in $\Sigma$ for $\neg\pi$
6: **else**
7:     $\sigma \leftarrow$ **create** or **retrieve existing** symbol in $\Sigma$ for $\pi$
8: **return** new automaton $\mathcal{A} = (\{q_1, q_2\}, \{(q_1, \sigma, q_2)\}, \{q_1\}, \{q_2\})$
```

**Algorithm 5.1:** Automaton algorithm for HDL Boolean expressions.

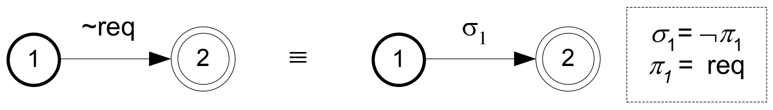


**Figure 5.1:** Automaton for Boolean $\sim req$, denoted $\mathcal{A}^B(\sim req)$, using Proposition 5.2. Simplified representation (left) and detailed representation (right).

where the dual-layer symbol alphabet takes form. The automaton that is returned in line 8 has two states, the first is an initial state and the second is a final state. A single edge connects the two in the proper direction. An example is shown in Figure 5.1 for the Boolean $\sim req$, where $req$ is a valid signal in the design to be verified. The simplified representation on the left will be used more often; however, the detailed representation shows how the symbol tables are actually used.

## 5.3 Implementation of Sequences

A sequence is converted to an equivalent automaton in an inductive manner. First, terminal automata are built for the Booleans in a sequence. Next, these automata are recursively combined according to the sequence operators comprising the given expression.

Sequences are an important part of the temporal layer in PSL, and are implemented using either rewrite rules or separate algorithms. Although no distinction is made here as to what operators are sugaring rules and which are base cases, as will be pointed out further, some of the rewrite rules are based on the sugaring definitions in Appendix B of the PSL specification [111]. Other sequence operators are implemented with specialized algorithms, such as fusion and intersection. These two operators are not often used in automata for conventional regular expressions, and the related algorithms will each be treated in their own subsection. An important factor that must be considered in these and other automata algorithms is the symbolic alphabet used to perform pattern matching over Boolean expressions.

The automaton implementation of sequences is shown below in Proposition 5.3. Items with an italicized comment in square brackets are part of the BNF syntax specification and were maintained to keep Proposition 5.3 as similar as possible to its related definition (Definition 2.8).

**Proposition 5.3:** If $i$ and $j$ represent nonnegative integers with $j \geq i$, and $k$ and $l$ represent positive integers with $l \geq k$, $b$ represents a Boolean, $s$ represents a Sequence, $r$ represents a SERE, $r\_r$ represents a repeated_SERE, $c\_r$ represents a compound_SERE, then the automaton implementation of *sequences*, denoted $\mathcal{A}^S(s)$, is performed as follows:

$\mathcal{A}^S(s)$ :

- $\mathcal{A}^S(r\_r) \quad = \quad$ [*defined below*]
- $\mathcal{A}^S(\{r\}) \quad = \quad \mathcal{A}^S(r)$

$\mathcal{A}^S(r)$ :

- $\mathcal{A}^S(b) \quad = \quad \mathcal{A}^B(b)$
- $\mathcal{A}^S(s) \quad = \quad$ [*defined above*]
- $\mathcal{A}^S(r_1 \ ; \ r_2) \quad = \quad \textsc{Concatenate}(\mathcal{A}^S(r_1), \mathcal{A}^S(r_2))$
- $\mathcal{A}^S(r_1 : r_2) \quad = \quad \textsc{Fuse}(\mathcal{A}^S(r_1), \mathcal{A}^S(r_2))$
- $\mathcal{A}^S(c\_r) \quad = \quad$ [*defined in next line*]

$\mathcal{A}^S(c\_r)$ :

- $\mathcal{A}^S(s) \quad = \quad$ [*defined above*]
- $\mathcal{A}^S(c\_r_1 \mid c\_r_2) \quad = \quad \textsc{Choice}(\mathcal{A}^S(c\_r_1), \mathcal{A}^S(c\_r_2))$
- $\mathcal{A}^S(c\_r_1 \ \& \ c\_r_2) \quad = \quad \mathcal{A}^S(\{\{c\_r_1\}\&\&\{c\_r_2;[*]\}\} \mid \{\{c\_r_1;[*]\}\&\&\{c\_r_2\}\})$
- $\mathcal{A}^S(c\_r_1 \ \&\& \ c\_r_2) \quad = \quad \textsc{Intersect}(\mathcal{A}^S(c\_r_1), \mathcal{A}^S(c\_r_2))$
- $\mathcal{A}^S(c\_r_1 \ \mathsf{within} \ c\_r_2) \quad = \quad \mathcal{A}^S(\{[*];c\_r_1;[*]\} \ \&\& \ \{c\_r_2\})$

$\mathcal{A}^S(r\_r)$ :

- $\mathcal{A}^S(b[*]) \quad = \quad \textsc{KleeneClosure}(\mathcal{A}^B(b))$
- $\mathcal{A}^S(b[*i]) \quad = \quad \mathcal{A}^S(b[*i{:}i])$
- $\mathcal{A}^S(b[*i{:}j]) \quad = \quad \textsc{RangeRepeat}(i, j, \mathcal{A}^B(b))$
- $\mathcal{A}^S(b[*i{:}\mathsf{inf}]) \quad = \quad \mathcal{A}^S(b[*i];b[*])$
- $\mathcal{A}^S(s[*]) \quad = \quad \textsc{KleeneClosure}(\mathcal{A}^S(s))$
- $\mathcal{A}^S(s[*i]) \quad = \quad \mathcal{A}^S(s[*i{:}i])$
- $\mathcal{A}^S(s[*i{:}j]) \quad = \quad \textsc{RangeRepeat}(i, j, \mathcal{A}^S(s))$
- $\mathcal{A}^S(s[*i{:}\mathsf{inf}]) \quad = \quad \mathcal{A}^S(s[*i];s[*])$
- $\mathcal{A}^S([*]) \quad = \quad \mathcal{A}^S(\mathsf{true}[*])$

- $\mathcal{A}^S$([*$i$]) $=$ $\mathcal{A}^S$(true[*$i$])
- $\mathcal{A}^S$([*$i$:$j$]) $=$ $\mathcal{A}^S$(true[*$i$:$j$])
- $\mathcal{A}^S$([*$i$:inf]) $=$ $\mathcal{A}^S$(true[*$i$:inf])
- $\mathcal{A}^S$($b$[+]) $=$ $\mathcal{A}^S$($b$;$b$[*])
- $\mathcal{A}^S$($s$[+]) $=$ $\mathcal{A}^S$($s$;$s$[*])
- $\mathcal{A}^S$([+]) $=$ $\mathcal{A}^S$(true; true[*])
- $\mathcal{A}^S$($b$[=$i$]) $=$ $\mathcal{A}^S$($b$[=$i$:$i$])
- $\mathcal{A}^S$($b$[=$i$:$j$]) $=$ $\mathcal{A}^S$({~$b$[*];$b$}[*$i$:$j$]; ~$b$[*])
- $\mathcal{A}^S$($b$[=$i$:inf]) $=$ $\mathcal{A}^S$($b$[=$i$]; [*])
- $\mathcal{A}^S$($b$[->]) $=$ $\mathcal{A}^S$($b$[->1])
- $\mathcal{A}^S$($b$[->$k$]) $=$ $\mathcal{A}^S$($b$[->$k$:$k$])
- $\mathcal{A}^S$($b$[->$k$:$l$]) $=$ $\mathcal{A}^S$({~$b$[*];$b$}[*$k$:$l$])
- $\mathcal{A}^S$($b$[->$k$:inf]) $=$ $\mathcal{A}^S$({$b$[->$k$]} ; {[*0]}|{[*];$b$})

The item $\mathcal{A}^S(b) = \mathcal{A}^B(b)$ uses the automata construction for Booleans defined in Proposition 5.2. All other items fall into two categories, namely those with direct algorithm implementations, and those that are implemented using rewrite rules. Those with rewrite rules can be easily identified since they have the form $\mathcal{A}^S() = \mathcal{A}^S()$, and will be treated in Section 5.3.5.

### 5.3.1 Conventional Regular Expression Operators

The conventional regular expression operators are concatenation, choice and Kleene closure (Kleene star), and are also used in sequences. The Boolean expression symbol alphabet produces nondeterministic automata because from a given state, two distinct symbols can cause simultaneous outgoing transitions when their respective Boolean expressions are both true. The NFA construction for conventional operators that was presented in Figure 2.5 can be reused given that both types of automata are nondeterministic. Since the automata for assertions (Definition 4.3) allow multiple initial states and do not use $\epsilon$ transitions, alternate algorithms for the three conventional operators must be developed.

The concatenation of two compound_SEREs is implemented as follows.

$$\mathcal{A}^S(r_1 \; ; \; r_2) \quad = \quad \textsc{Concatenate}(\mathcal{A}^S(r_1),\, \mathcal{A}^S(r_2))$$

Algorithm 5.2 presents the algorithm used for the concatenation of two argument automata. In concatenation, for each final state in the left-side automaton, a copy of

```
1: FUNCTION: CONCATENATE($\mathcal{A}_L$, $\mathcal{A}_R$)
2: **create** new automaton $\mathcal{A} = (Q_L \cup Q_R, \delta_L \cup \delta_R, I_L, F_R)$   $//Q, \delta, I, F$
3: **for** each state $f_L \in F_L$ **do**
4:    **for** each edge $(s_R, \sigma_R, d_R) \in \delta_R \mid s_R \in I_R$ **do**
5:       $\delta \leftarrow \delta \cup \{(f_L, \sigma_R, d_R)\}$
6:       **if** $s_R \in F_R$ **then**
7:          $F \leftarrow F \cup \{f_L\}$
8: **return** $\mathcal{A}$
```

**Algorithm 5.2:** Automata concatenation algorithm.

```
1: FUNCTION: CHOICE($\mathcal{A}_1$, $\mathcal{A}_2$)
2: **return** new automaton $\mathcal{A} = (Q_L \cup Q_R, \delta_L \cup \delta_R, I_L \cup I_R, F_L \cup F_R)$
```

**Algorithm 5.3:** Automata choice algorithm.

all edges that originate from the initial states in the right-side automaton is made, whereby the new edges originate from the given final state in the left side (lines 3 to 5). In this manner, each time the left side is matched, the right side is triggered. The final states correspond to the right-side automaton's final states (line 2). When the right side has an initial state that is also a final state, final states in the left side are kept final so that the left side can also cause a successful match (lines 6 and 7).

The disjunction of two compound_SEREs is similar to the choice operation of conventional regular expressions, and is implemented as follows.

$$\mathcal{A}^S(c\_r_1 \mid c\_r_2) \quad = \quad \textsc{Choice}(\mathcal{A}^S(c\_r_1),\, \mathcal{A}^S(c\_r_2))$$

Automata choice is rather straightforward, given that multiple initial states are permitted by the automaton definition (Definition 4.3). The algorithm for performing automata choice is presented in Algorithm 5.3. With multiple initial states, automata choice amounts to integrating both argument automata into the same result automaton, such that both are disjoint, and the set of states, the set of transitions, the set of initial states and the set of final states are merged (line 2).

Two items in Proposition 5.3 make use of Kleene closure:

$$\begin{aligned}\mathcal{A}^S(b[*]) \quad &= \quad \textsc{KleeneClosure}(\mathcal{A}^B(b))\\ \mathcal{A}^S(s[*]) \quad &= \quad \textsc{KleeneClosure}(\mathcal{A}^S(s))\end{aligned}$$

The Kleene closure (or star repetition) of a Sequence or Boolean is performed using the KLEENECLOSURE() algorithm, presented in Algorithm 5.4. Each edge that hits a

```
1: FUNCTION: KleeneClosure($\mathcal{A}$)
2: create transition relation $\delta_1 = \emptyset$
3: for each edge $(s, \sigma, d) \in \delta \mid d \in F$ do
4:     for each state $i \in I$ do
5:         $\delta_1 \leftarrow \delta_1 \cup \{(s, \sigma, i)\}$
6: $\delta \leftarrow \delta \cup \delta_1$
7: $F \leftarrow F \cup I$
8: return $\mathcal{A}$
```

**Algorithm 5.4:** Automaton Kleene-closure algorithm.

final state is replicated using the same symbol, whereby the destination states of the new edges correspond to the initial states (lines 3 to 5). Thus, each time a final state is activated, the automaton is also automatically retriggered. In line 7, the initial states are made final states, such that the empty match is performed as required for Kleene closure. The transition relation $\delta_1$ used in lines 2, 5 and 6 ensures that the algorithm operates correctly and does not enter a potentially infinite loop.

All three algorithms for the concatenation, choice and Kleene closure operators are linear in the size (number of states) of the input automata. An indication that these algorithms are not restricted to automata that use the two-layer symbol alphabet lies in the observation that no symbol sets ($\Pi$ nor $\Sigma$) are affected by these algorithms, and they could well be used in the context of conventional regular expressions.

Examples for the three conventional operators are shown in Figure 5.2. The automata for the empty and null sequences (Definition 2.9) are also illustrated to show the nuance with how they were defined in Figure 2.5 for conventional NFAs. The automaton for matching the empty sequence consists of a single state which is both an initial state and a final state. This automaton is produced when the sequence [*0] is used. The automaton for matching the null sequence consists in a single initial state with no final states, and is a valid automaton according to Definition 4.3. This automaton is produced when the sequence {[*1]&&[*2]} is used, for example.

### 5.3.2 Fusion

In Proposition 5.3, the fusion of two SEREs was implemented as follows.

$$\mathcal{A}^S(r_1 : r_2) \quad = \quad \textsc{Fuse}(\mathcal{A}^S(r_1), \mathcal{A}^S(r_2))$$

As observed in Section 2.4, SERE fusion corresponds to an overlapped concatenation. The algorithm for performing the fusion of two automata is presented in Al-

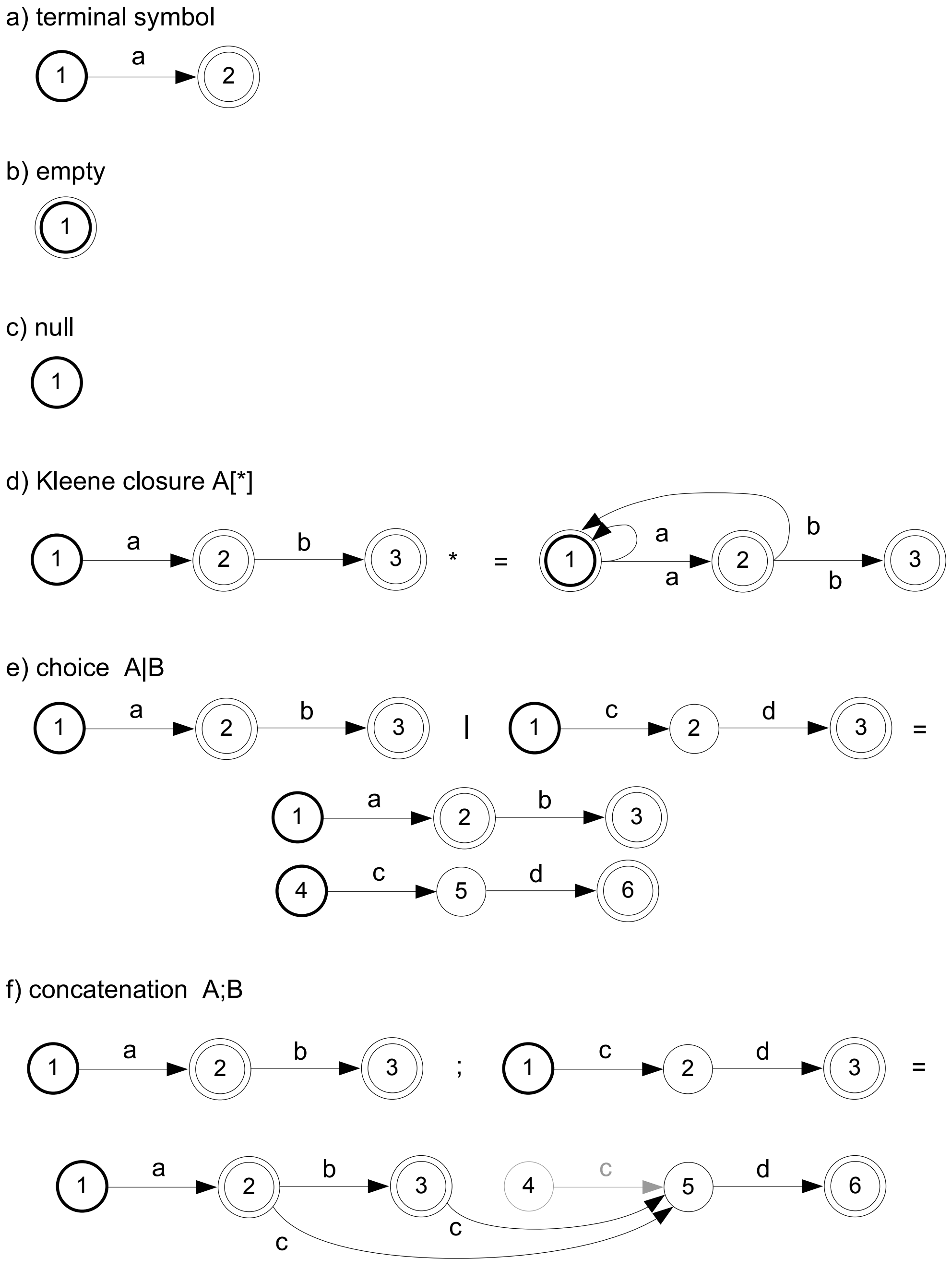


**Figure 5.2:** Automata examples for conventional RE operators. In the examples, $a$, $b$, $c$ and $d$ are Booleans, automaton $A$ matches the sequence $\{a;b[*0:1]\}$ and automaton $B$ matches the sequence $\{c;d\}$. In the concatenation example, the state in light-gray is pruned during minimization.

```
1: FUNCTION: FUSE($\mathcal{A}_L$, $\mathcal{A}_R$)
2: create new automaton $\mathcal{A} = (Q_L \cup Q_R, \delta_L \cup \delta_R, I_L, F_R)$   $//Q, \delta, I, F$
3: for each edge $(s_L, \sigma_L, d_L) \in \delta_L \mid d_L \in F_L$ do
4:    for each edge $(s_R, \sigma_R, d_R) \in \delta_R \mid s_R \in I_R$ do
5:       $\sigma \leftarrow$ create or retrieve symbol in $\Sigma$ for $\sigma_L \wedge \sigma_R$
6:       if EOE signal $\notin$ $\sigma$'s primary symbols then
7:          $\delta \leftarrow \delta \cup \{(s_L, \sigma, d_R)\}$
8: return $\mathcal{A}$
```

**Algorithm 5.5:** Automata fusion algorithm.

gorithm 5.5. The algorithm starts by incorporating both argument automata into a new automaton such that they are disjoint (line 2). From there, intersection edges are created from edges that hit final states in the left-side automaton and edges that leave the initial states in the right-side automaton. The intersection of symbols in line 5 benefits from the polarity that is encoded in the dual symbol tables, and can detect conditions where Boolean conjunctions simplify to false. In such cases no edge is actually created in line 7 (not shown in the algorithm). The extended symbol created or retrieved in line 5 does not affect the set of primary symbols Π. The condition in line 6 has no effect and is always true in SEREs because the End of Execution (EOE) signal applies only at the property level. This line is required to ensure the proper behavior of certain strong properties that also make use of the fusion algorithm, and is explained in Section 5.4.1.

An illustration of SERE fusion is shown in Figure 5.3 for the SERE $\{a;b[*]\}:\{c;d\}$. SERE fusion produces an automaton that has $O(m+n)$ states, where $m$ and $n$ are the sizes of the input automata. Because empty SEREs on either side do not result in a match, final states in the left-side automaton are not final states in the resulting automaton. Similarly, the initial states in the right-side automaton are not initial states in the resulting automaton (line 2). Thus, if an initial state in the right side is a final state, it will not result in an instant match. The number of edges that are added can be determined by examining the two **for** loops in the algorithm.

### 5.3.3 Length-Matching Intersection

The length-matching intersection of two compound_SEREs was implemented as follows in Proposition 5.3.

$$\mathcal{A}^S(c\_r_1 \;\&\&\; c\_r_2) \quad = \quad \textsc{Intersect}(\mathcal{A}^S(c\_r_1),\, \mathcal{A}^S(c\_r_2))$$

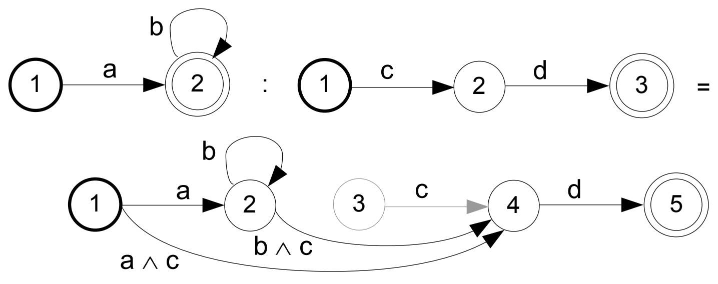


**Figure 5.3:** SERE fusion example for $\{a;b[*]\}$:$\{c;d\}$. The state in light-gray is pruned during minimization.

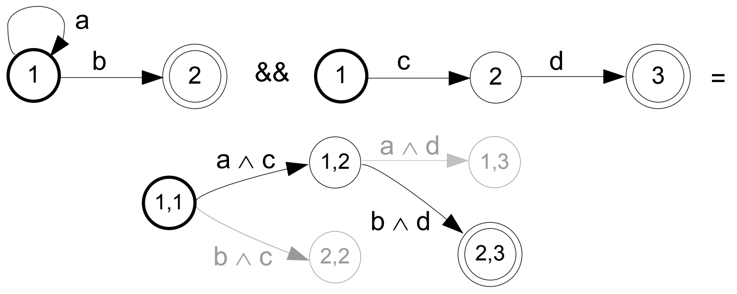


**Figure 5.4:** Sequence intersection example for $\{a[*];b\}$&&$\{c;d\}$. States in light-gray are pruned.

Typical automata intersection [101], which equates to building a *product automaton* of both arguments, is incompatible with the symbolic alphabet and can not be used directly in the intersection algorithm. If the traditional intersection procedure is applied to two automata that have no symbol in common (i.e. no syntactically equal symbols), an empty automaton results. This automaton can obviously not detect the intersection of two SEREs which use disjoint sets of extended symbols.

To implement the intersection algorithm correctly, the condition on syntactic equality of symbols must be relaxed and the conjunction of symbols must be considered by the algorithm. To consider all relevant pairs of edges, the product automaton is constructed in such a way as to perform Boolean intersection between pairs of edges, where each edge in a pair is from a separate automaton. As shown in the intersection algorithm in Algorithm 5.6, a state in the intersection automaton is labeled by an ordered pair $(u, v)$, where $u$ and $v$ correspond to states from the first and second argument automata respectively.

The *state creation set* (line 8) is the key component in the algorithm, and is initialized to the pair of initial states (line 6) of both argument automata. The two input automata are weakly determinized in lines 2 and 3; a side-effect of this is that

```
1: FUNCTION: INTERSECT($\mathcal{A}_1$, $\mathcal{A}_2$)
2: $\mathcal{A}_1 \leftarrow$ WEAKDETERMINIZE($\mathcal{A}_1$);
3: $\mathcal{A}_2 \leftarrow$ WEAKDETERMINIZE($\mathcal{A}_2$);
4: //here $|I_1|$=$|I_2|$=1; i.e. $I_1 = \{i_1\}$, $I_2 = \{i_2\}$
5: //a label is an ordered pair $(u, v) \mid u \in Q_1,\ v \in Q_2$
6: create new state $q$ labeled with $(i_1, i_2)$
7: create new automaton $\mathcal{A} = (\{q\}, \emptyset, \{q\}, \emptyset)$    $//Q, \delta, I, F$
8: add $q$ to an initially empty set $C$   $//C$ is "to-construct"
9: while $C \neq \emptyset$ do
10:    remove a state $r$ (with its label $(u, v)$), from $C$
11:    if $u \in F_1$ and $v \in F_2$ then
12:       $F \leftarrow F \cup \{r\}$
13:    for each edge $(s_1, \sigma_1, d_1) \in \delta_1 \mid s_1 = u$ do
14:       for each edge $(s_2, \sigma_2, d_2) \in \delta_2 \mid s_2 = v$ do
15:          find state $t \in Q$ with label $(d_1, d_2)$
16:          if $\nexists t$ then
17:             create new state $t \in Q$ labeled with $(d_1, d_2)$
18:             $C \leftarrow C \cup \{t\}$
19:          $\sigma \leftarrow$ create or retrieve symbol in $\Sigma$ for $\sigma_1 \wedge \sigma_2$
20:          $\delta \leftarrow \delta \cup \{(r, \sigma, t)\}$
21: return $\mathcal{A}$
```

**Algorithm 5.6:** Automata intersection algorithm.

each resulting automaton has a single initial state. The automata are made weakly deterministic so that when considering a symbol from an edge in each automaton, $\sigma_1$ and $\sigma_2$ for instance, there is one and only one destination state in each automaton, such that a state pair $(d_1, d_2)$ represents a new state in the intersection automaton. The two **for** loops in lines 13 and 14 intersect two states, and the **while** loop together with the set $C$ create the entire intersection automaton. A new state $(u, v)$ is a final state if and only if states $u$ and $v$ are final states in their respective automata. The symbol created or retrieved in line 19 affects only the set of extended symbols $\Sigma$.

The algorithm creates new states and edges only for the reachable states of the resulting automaton. The symbol intersection in line 19 also benefits from the polarity encoded in the dual symbol tables, and can detect conditions where Boolean conjunctions simplify to false. In such cases no edge is created, and the block of statements in lines 15 to 20 is not executed (not shown in the algorithm).

An example depicting intersection is shown in Figure 5.4 for {*a*[*];*b*}&&{*c*;*d*}. Sequence intersection produces an automaton that has in the worst case $O(mn)$ states, where $m$ and $n$ are the number of states in the input automata. The algorithm is proven to terminate because only a finite number of states can be added to the

```
1: FUNCTION: RangeRepeat($low$, $high$, $\mathcal{A}$)
2: if $high = 0$ then    $//high \geq low$ is assumed
3:    create new automaton $\mathcal{A}_1 = (\{q_1\}, \emptyset, \{q_1\}, \{q_1\})$    $//Q_1, \delta_1, I_1, F_1$
4: else
5:    create new automaton $\mathcal{A}_1 = \mathcal{A}$
6:    for $i \leftarrow 2$ to $high$ do
7:       $\mathcal{A}_1 \leftarrow$ Concatenate($\mathcal{A}_1$, $\mathcal{A}$)
8:       if $i \leq (high - low + 1)$ then
9:          $I_1 \leftarrow I_1 \cup \{s \mid s \in Q_1$, $s$ was in $I$ (in $\mathcal{A}$) before the concatenation$\}$
10:   if $low = 0$ then
11:      $F_1 \leftarrow F_1 \cup I_1$
12: return $\mathcal{A}_1$
```

**Algorithm 5.7:** Automata range repetition algorithm.

construction set.

### 5.3.4 Repetition with a Range

Sequence and Boolean repetition with a range is required to implement the two such items from Proposition 5.3:

$$\begin{aligned}\mathcal{A}^S(b[*i{:}j]) &= \textsc{RangeRepeat}(i,\ j,\ \mathcal{A}^B(b))\\ \mathcal{A}^S(s[*i{:}j]) &= \textsc{RangeRepeat}(i,\ j,\ \mathcal{A}^S(s))\end{aligned}$$

The PSL specification indicates that sequence repetition with a range can be defined using sequence disjunction as follows:

$$s[*i{:}j] \quad \overset{\text{def}}{=} \quad s[*i] \mid \ldots \mid s[*j] \tag{5.2}$$

The above equation could be used to form a rewrite rule in the next section; however, since the actual syntactic size of the expression on the right side is not fixed in size and is a function of the parameters $i$ and $j$, it is more efficient to develop a particular algorithm for the range repetition, both in the interest of efficient rewriting and for the actual creation of automata. Whether for a Boolean or a sequence, the range repetition of an automaton is performed using the algorithm in Algorithm 5.7.

The algorithm assumes that the upper bound for the repetition is greater or equal to the lower bound. The special case $high = 0$ is handled separately in line 3, and produces the empty automaton that was shown in Figure 5.2 b). When $high > 0$, the returned automaton is initially formed as a copy of the argument automaton (line 5). Lines 6 and 7 concatenate additional copies of the argument automaton $high-1$ times (line 6: **for** loop from 2 to $high$). When $high < 2$, this block of lines has no effect since no concatenations are needed. Normally, concatenation does not include the initial states of the right-side automaton in the set of initial states of the resulting automaton. Lines 8 and 9 counteract this and are the mechanism by which the repetition *range* is implemented. The first $high - low + 1$ concatenations' initial states will remain as initial states for the resulting automaton. This has the effect of triggering the concatenation chain for a certain prefix of copies of the concatenated argument automata, such that different lengths of matching can be produced. This technique is referred to as *initial-state prefix triggering*.

Another technique for implementing range repetition consists in instead adding the final states of the left-side automaton in the concatenation step to the set of final states when $i \geq low$. This is not optimal as the further concatenations in the loop involve progressively more and more final states from the growing left side automaton, and hence more edges are generated. Even though minimization can reduce the size of the automaton, the algorithm should strive to be the most efficient possible especially when a large range is used. This technique is referred to as *final-state suffix signaling*.

The two range repetition techniques introduced above are illustrated in Figure 5.5. In this figure the concatenation symbols are meant to indicate that concatenations *were* performed, and for simplicity no edges were actually represented. The focus of this figure is on the initial states and the final states that remain after the concatenations have been performed.

In both techniques, the resultant automata match anywhere from two to five instances of the argument automaton $A$. The only disadvantage is that with *final-state suffix signaling* (at the bottom of the figure), concatenations are performed on a left-side automaton that has more and more final states, and a growing number of edges come into play. For example, if a sixth instance of automaton $A$ is to be concatenated at the end of the graphs in Figure 5.5, the concatenation algorithm in Algorithm 5.2 will cause many more edges to be added in part b) than in part a). The range repetition algorithm creates an automaton that is linear in size complexity with respect to the size (number of states) of the argument automaton.

### 5.3.5 Rewrite Rules

The items in Proposition 5.3 that have not been treated thus far have a common characteristic: they have the form $\mathcal{A}^S(x) = \mathcal{A}^S(y)$. More generally, the link between

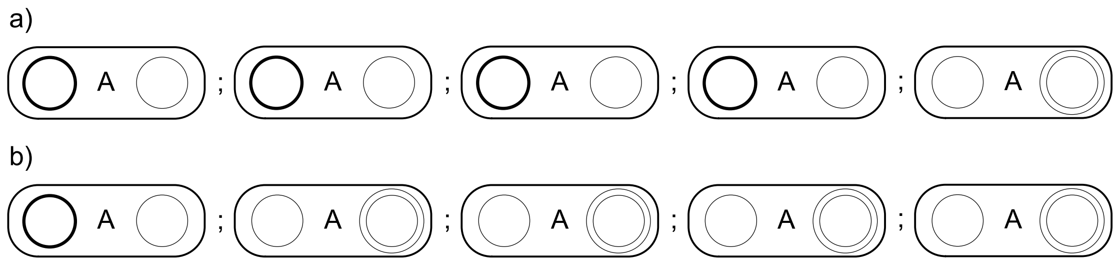


**Figure 5.5:** Range repetition strategies for: a) initial-state prefix triggering and b) final-state suffix signaling. The automaton $A$ was repeated with the range $low = 2$, $high = 5$.

the arguments can be expressed as a rewrite rule of the form $x \rightarrow y$, where $x$ is rewritten to $y$. Expression rewriting (or term rewriting) is a practical way of allowing the checker generator to handle the large amount of sequence operators, while actually only implementing algorithms for a much smaller number of operators in the tool's kernel.

The rewrite rules used are either derived from the SERE sugaring definitions in Appendix B in the PSL specification [111] directly, or with small modifications. Although a few rewrite rules may appear intuitive, they are nonetheless included for completeness. Each rule is explained immediately following its appearance.

$$\{r\} \quad \rightarrow \quad r$$

The above rule is straightforward, given that semantically the curly brackets are used only for grouping and their main effect is instead in the syntax of SEREs.

$$c\_r_1 \text{ \& } c\_r_2 \quad \rightarrow \quad \{\{c\_r_1\}\&\&\{c\_r_2;[*]\}\} \,|\, \{\{c\_r_1;[*]\}\&\&\{c\_r_2\}\}$$

The preceding rule for non-length matching SERE intersection attempts to extend the shortest of the two SEREs with a [*] in order to use length matching intersection. Since the shorter of the two SEREs can not be known *a priori*, the two possibilities are modeled and connected with SERE disjunction.

$$c\_r_1 \text{ within } c\_r_2 \quad \rightarrow \quad \{[*];c\_r_1;[*]\} \;\&\&\; \{c\_r_2\}$$

When a compound SERE must be matched within the matching of another, the first SERE is made to start at a varying point with respect to the second SERE by concatenating a leading [*]. The SERE is also lengthened with a trailing [*] so that mapping to length matching intersection can be performed. The within rewrite and the non-length matching intersection rewrite (&) both follow directly from the PSL specification [111].

$$\begin{aligned} b[*i] &\rightarrow b[*i{:}i] \\ s[*i] &\rightarrow s[*i{:}i] \end{aligned}$$

The single number repetition, whether for Booleans or sequences, is rewritten to the range repetition given that the range repetition must also be implemented.

$$\begin{aligned} b[*i{:}\mathsf{inf}] &\rightarrow b[*i];b[*] \\ s[*i{:}\mathsf{inf}] &\rightarrow s[*i];s[*] \end{aligned}$$

When the upper bound in a range repetition is infinite, the sequence or Boolean is repeated the minimum number of times, and is subsequently allowed to be matched any additional number of times (including 0 times), hence the concatenated Kleene closure of the sequence or Boolean.

$$\begin{aligned} [*] &\rightarrow \mathsf{true}[*] \\ [*i] &\rightarrow \mathsf{true}[*i] \\ [*i{:}j] &\rightarrow \mathsf{true}[*i{:}j] \\ [*i{:}\mathsf{inf}] &\rightarrow \mathsf{true}[*i{:}\mathsf{inf}] \end{aligned}$$

The various forms of [*] repetition that do not specify a sequence or Boolean implicitly apply to the Boolean true, and can be rewritten as such, as shown above.

$$\begin{aligned} b[+] &\rightarrow b;b[*] \\ s[+] &\rightarrow s;s[*] \\ [+] &\rightarrow \mathsf{true};\ \mathsf{true}[*] \end{aligned}$$

The various forms of [+] repetition above correspond to repetitions of one or more instances, and can be rewritten using [*] repetition. The three previous groups of rewrites follow directly from the PSL specification [111].

$$\begin{aligned} b[->] &\rightarrow b[->1] \\ b[->k] &\rightarrow b[->k{:}k] \end{aligned}$$

Both forms of goto repetition above are rewritten to more complex forms of goto repetition. The goto repetitions are actually defined differently in the PSL specification [111], as shown below:

$$b[\text{->}] \quad \overset{\text{def}}{=} \quad \sim b[*];b \tag{5.3}$$

$$b[\text{->}k] \quad \overset{\text{def}}{=} \quad \{\sim b[*];b\}[*k] \tag{5.4}$$

The form chosen in this work is equivalent to the one in the PSL specification, and more closely reflects how those operators are actually implemented in the checker generator. In some cases, mapping an operator to a more complex variant of the same operator can be a simpler solution.

$$b[\text{->}k{:}l] \quad \rightarrow \quad \{\sim b[*];b\}[*k{:}l] \tag{5.5}$$

The goto repetition with a range is the most complex of goto repetitions, and is rewritten using a different approach than in the PSL specification. In the PSL specification [111], this operator is defined as:

$$b[\text{->}k{:}l] \quad \overset{\text{def}}{=} \quad b[\text{->}k] \mid \ldots \mid b[\text{->}l]$$

The expression on the right side has the disadvantage that its size depends on the values of $k$ and $l$. The form chosen in (5.5) is more efficient for term rewriting because of its fixed size. The following proof shows the equivalency of the version in the PSL specification [111] with the version used in (5.5):

$$\begin{array}{ll} & b[\text{->}k] \mid \ldots \mid b[\text{->}l] \\ \Leftrightarrow & \{\sim b[*];b\}[*k] \mid \ldots \mid \{\sim b[*];b\}[*l] \qquad //\text{using (5.4)} \\ \Leftrightarrow & \{\sim b[*];b\}[*k{:}l] \qquad //\text{using (5.2); Q.E.D.} \end{array}$$

All rewrite rules developed in this work do not have variable right-hand sizes, and a more static form is preferred for performance reasons and for simplicity of implementation in the checker generator.

It should also be stated that when devising rewrite rules, proper care must be taken to ensure that the set of rewrite rules is *terminating* (Definition 2.13). A non-terminating set of rewrite rules runs the risk of entering an infinite substitution cycle, which is obviously not desired. For example, if the right side in (5.5) was $\{b[\text{->}]\}[*k{:}l]$, the set would be non-terminating. Here is how the non-terminating, thus *incorrect* set of goto rules would appear:

$$\begin{aligned} b[\texttt{->}] &\quad\rightarrow\quad b[\texttt{->}1] \\ b[\texttt{->}k] &\quad\rightarrow\quad [\texttt{->}k{:}k] \\ b[\texttt{->}k{:}l] &\quad\rightarrow\quad \{b[\texttt{->}]\}[*k{:}l] \end{aligned}$$

In the above rules, there is an infinite loop as the first rule uses the second, the second uses the third, and the third rule uses the first.

$$b[\texttt{->}k{:}\mathsf{inf}] \quad\rightarrow\quad \{b[\texttt{->}k]\}\ ;\ \{[*0]\}|\{[*];b\} \tag{5.6}$$

Goto repetition with an infinite upper bound is rewritten above in a slightly different manner than the definition in the PSL specification [111]. The PSL specification has the following expression as the right side:

$$b[\texttt{->}k{:}\mathsf{inf}] \quad\overset{\text{def}}{=}\quad b[\texttt{->}k]\ |\ \{b[\texttt{->}k];[*];b\} \tag{5.7}$$

The forms in (5.7) in (5.6) are equivalent because the expression $b[\texttt{->}k]$ can be *temporally factored*. Temporally factoring a sequence corresponds to removing a common temporal prefix in two subsequences, and concatenating it with the reduced subsequences. In the case used above, the temporal factoring of $b[\texttt{->}k]$ from the disjunction requires the use of the empty SERE, and yields the form in (5.6). The factored form was chosen for efficiency reasons to avoid building an automaton for $b[\texttt{->}k]$ twice. Temporal factoring can also apply to common suffixes in sequences.

$$b[=i] \quad\rightarrow\quad b[=i{:}i] \tag{5.8}$$

The single parameter non-consecutive repetition shown above is rewritten to a more complex form of the same operator. This operator is defined differently in the PSL specification [111]:

$$b[=i] \quad\overset{\text{def}}{=}\quad \{\sim b[*];b\}[*i];\sim b \tag{5.9}$$

The approach chosen for this operator was also to map it to a more complex version of the same operator.

$$b[=i{:}j] \quad\rightarrow\quad \{\sim b[*];b\}[*i{:}j];\ \sim b[*] \tag{5.10}$$

Non-consecutive repetition with a range, as shown above, is implemented with a rewrite that differs from the PSL specification [111]. The documented definition of non-consecutive repetition with a range is:

$$b[=i{:}j] \quad \overset{\text{def}}{=} \quad b[=i] \mid \ldots \mid b[=j]$$

The expression on the right side once again has the disadvantage that its size depends on the values of the parameters. Similarly to the goto repetition case, the form chosen in (5.10) is more efficient for term rewriting because of its fixed size. The following proof shows the equivalency of the definition in the PSL specification [111] with the version used in (5.10):

$$\begin{array}{ll} & b[=i] \mid \ldots \mid b[=j] \\ \Leftrightarrow & \{\sim b[*];b\}[*i];\sim b \mid \ldots \mid \{\sim b[*];b\}[*j];\sim b \qquad //\text{using (5.9)} \\ \Leftrightarrow & \{\ \{\sim b[*];b\}[*i] \mid \ldots \mid \{\sim b[*];b\}[*j]\ \}\ ;\ \sim b \qquad //\text{temporal suffix factoring} \\ \Leftrightarrow & \{\sim b[*];b\}[*i{:}j]\ ;\ \sim b \qquad //\text{using (5.2); Q.E.D.} \end{array}$$

The above derivation is used to create a rewrite rule with a fixed size on the right side.

$$b[=i{:}\mathsf{inf}] \quad = \quad b[=i];\ [*]$$

The final rewrite rule left to cover is the goto repetition with an infinite upper bound, which follows directly from its definition in the PSL specification [111].

This concludes the implementation of rewrite rules for sequences, and together with the algorithms from the previous subsections, automata for recognizing PSL sequences can be constructed. Sequences form an important part of properties, which are implemented next. Surprisingly, many algorithms devised for sequence operators are also reused at the property level. Rewrite rules also play a very important role for handling the large amount of property operators.

## 5.4 Implementation of Properties

Continuing the bottom-up construction of PSL directives now leads to the implementation of properties. The automaton for a property $p$ is denoted $\mathcal{A}^P(p)$. The superscript $^P$ is used to indicate the context of properties. It does not represent a new automaton type, and is used to remove the ambiguity between the automaton for a Boolean used as a property and the automaton for a plain Boolean, from Section 5.2. An automaton $\mathcal{A}^P$ is defined exactly the same way as an automaton $\mathcal{A}$

from the previous chapter, and all automata algorithms from that chapter can be used here with $\mathcal{A}^P$ automata as arguments. The superscript is particularly important in properties because $\mathcal{A}^P(b) \neq \mathcal{A}^B(b)$ : the left side builds an automaton from a Boolean $b$ appearing as property, whereas the right side builds an automaton for a plain Boolean, appearing in a Sequence for example.

The automaton implementation of properties is presented below in Proposition 5.4, in a form that resembles the definition of properties (Definition 2.10) as most as possible, so that the link between syntax and implementation can be better observed.

**Proposition 5.4:** If $i$ and $j$ represent nonnegative integers with $j \geq i$, $k$ and $l$ represent positive integers with $l \geq k$, $b$ is a Boolean, $s$ is a Sequence, $f\_p$ is a FL_Property, $p$ is a property, then the automaton implementation of *properties*, denoted $\mathcal{A}^P(p)$, is performed as follows:

$\mathcal{A}^P(p)$ :

- $\mathcal{A}^P(\textsf{forall } ident \textsf{ in boolean: } p) \ = \ \textsc{Choice}(\mathcal{A}^P((p)_{ident \leftarrow \textsf{true}}), \mathcal{A}^P((p)_{ident \leftarrow \textsf{false}}))$
- $\mathcal{A}^P(\textsf{forall } ident \textsf{ in } \{i{:}j\}\textsf{: } p) \ = \ \textsc{ForallRange}(i, j, ident, p)$
- $\mathcal{A}^P(f\_p) \ = \ [\textit{defined in next line}]$

$\mathcal{A}^P(f\_p)$ :

- $\mathcal{A}^P(b) \ = \ \textsc{FirstFail}(\mathcal{A}^B(b))$
- $\mathcal{A}^P((f\_p)) \ = \ \mathcal{A}^P(f\_p)$
- $\mathcal{A}^P(s\,!) \ = \ \textsc{FirstFailStrong}(\mathcal{A}^S(s))$
- $\mathcal{A}^P(s) \ = \ \textsc{FirstFail}(\mathcal{A}^S(s))$
- $\mathcal{A}^P(f\_p \textsf{ abort } b) \ = \ \textsc{AddLiteral}(\mathcal{A}^P(f\_p), \mathcal{H}(\sim b))$
- $\mathcal{A}^P(!\,b) \ = \ \textsc{FirstFail}(\mathcal{A}^B(\sim b))$
- $\mathcal{A}^P(f\_p_1 \textsf{ \&\& } f\_p_2) \ = \ \textsc{Choice}(\mathcal{A}^P(f\_p_1), \mathcal{A}^P(f\_p_2))$
- $\mathcal{A}^P(b \mid\mid f\_p) \ = \ \mathcal{A}^P(\{\sim b\} \textsf{ |-> } f\_p)$
- $\mathcal{A}^P(b \textsf{ -> } f\_p) \ = \ \mathcal{A}^P(\{b\} \textsf{ |-> } f\_p)$
- $\mathcal{A}^P(b_1 \textsf{ <-> } b_2) \ = \ \textsc{FirstFail}(\mathcal{A}^B(b_1 \textsf{ <-> } b_2))$
- $\mathcal{A}^P(\textsf{always } f\_p) \ = \ \mathcal{A}^P(\{[+]\} \textsf{ |-> } f\_p)$
- $\mathcal{A}^P(\textsf{never } s) \ = \ \mathcal{A}^P(\{[+] : s\} \textsf{ |-> false})$
- $\mathcal{A}^P(\textsf{next } f\_p) \ = \ \mathcal{A}^P(\textsf{next}[1](f\_p))$
- $\mathcal{A}^P(\textsf{next! } f\_p) \ = \ \mathcal{A}^P(\textsf{next!}[1](f\_p))$
- $\mathcal{A}^P(\textsf{eventually! } s) \ = \ \mathcal{A}^P(\{[+] : s\}!)$
- $\mathcal{A}^P(f\_p \textsf{ until! } b) \ = \ \mathcal{A}^P((f\_p \textsf{ until } b) \textsf{ \&\& } (\{b[\textsf{->}]\}!))$
- $\mathcal{A}^P(f\_p \textsf{ until } b) \ = \ \mathcal{A}^P(\{(\sim b)[+]\} \textsf{ |-> } f\_p)$
- $\mathcal{A}^P(f\_p \textsf{ until!\_ } b) \ = \ \mathcal{A}^P((f\_p \textsf{ until\_ } b) \textsf{ \&\& } (\{b[\textsf{->}]\}!))$
- $\mathcal{A}^P(f\_p \textsf{ until\_ } b) \ = \ \mathcal{A}^P(\{\,\{(\sim b)[+]\} \mid \{b[\textsf{->}]\}\,\} \textsf{ |-> } f\_p)$

- $\mathcal{A}^P(b_1$ before! $b_2) = \mathcal{A}^P(\{(\sim b_1\&\sim b_2)$[*] ; $(b_1\&\sim b_2)\}!)$
- $\mathcal{A}^P(b_1$ before $b_2) = \mathcal{A}^P(\{(\sim b_1\&\sim b_2)$[*] ; $(b_1\&\sim b_2)\})$
- $\mathcal{A}^P(b_1$ before!_ $b_2) = \mathcal{A}^P(\{(\sim b_1\&\sim b_2)$[*] ; $b_1\}!)$
- $\mathcal{A}^P(b_1$ before_ $b_2) = \mathcal{A}^P(\{(\sim b_1\&\sim b_2)$[*] ; $b_1\})$
- $\mathcal{A}^P($next$[i](f_p)) = \mathcal{A}^P($next_event(true)$[i+1](f_p))$
- $\mathcal{A}^P($next!$[i](f_p)) = \mathcal{A}^P($next_event!(true)$[i+1](f_p))$
- $\mathcal{A}^P($next_a$[i{:}j](f_p)) = \mathcal{A}^P($next_event_a(true)$[i+1:j+1](f_p))$
- $\mathcal{A}^P($next_a!$[i{:}j](f_p)) = \mathcal{A}^P($next_event_a!(true)$[i+1:j+1](f_p))$
- $\mathcal{A}^P($next_e$[i{:}j](b)) = \mathcal{A}^P($next_event_e(true)$[i+1:j+1](b))$
- $\mathcal{A}^P($next_e!$[i{:}j](b)) = \mathcal{A}^P($next_event_e!(true)$[i+1:j+1](b))$
- $\mathcal{A}^P($next_event!$(b)(f_p)) = \mathcal{A}^P($next_event!$(b)[1](f_p))$
- $\mathcal{A}^P($next_event$(b)(f_p)) = \mathcal{A}^P($next_event$(b)[1](f_p))$
- $\mathcal{A}^P($next_event!$(b)[k](f_p)) = \mathcal{A}^P($next_event_a!$(b)[k{:}k](f_p))$
- $\mathcal{A}^P($next_event$(b)[k](f_p)) = \mathcal{A}^P($next_event_a$(b)[k{:}k](f_p))$
- $\mathcal{A}^P($next_event_a!$(b)[k{:}l](f_p)) =$
  $\mathcal{A}^P($next_event_a$(b)[k{:}l](f_p)$ && $\{b$[->$l]\}!)$
- $\mathcal{A}^P($next_event_a$(b)[k{:}l](f_p)) = \mathcal{A}^P(\{b$[->$k{:}l]\}$ |-> $f\_p)$
- $\mathcal{A}^P($next_event_e!$(b_1)[k{:}l](b_2)) = \mathcal{A}^P(\{b_1$[->$k{:}l]$ : $b_2\}!)$
- $\mathcal{A}^P($next_event_e$(b_1)[k{:}l](b_2)) = \mathcal{A}^P(\{b_1$[->$k{:}l]$ : $b_2\})$
- $\mathcal{A}^P(s$ |-> $f\_p) = \textsc{Fuse}(\mathcal{A}^S(s), \mathcal{A}^P(f\_p))$
- $\mathcal{A}^P(s$ |=> $f\_p) = \mathcal{A}^P(\{s$ ; true$\}$ |-> $f\_p)$

The following example motivates the importance of the superscript symbols in the automata construction formalism.

$$\mathcal{A}^P(s_1 \text{ |-> } s_2)$$

In the expression above, $s_1$ and $s_2$ are sequences; however, $s_2$ is used directly as a property whereas $s_1$ is the antecedent of a suffix implication (which must be a sequence proper). After one step of processing using the results above, the property reduces to:

$$\textsc{Fuse}(\mathcal{A}^S(s_1), \mathcal{A}^P(s_2))$$

Here it is apparent that $s_1$ and $s_2$ are not interpreted in the same manner (superscripts). Continuing one step further:

$$\textsc{Fuse}(\mathcal{A}^S(s_1), \textsc{FirstFail}(\mathcal{A}^S(s_2)))$$

Now both sequences can be constructed using the procedures in the previous section.

Many properties are implemented by either directly expressing them as sequence expressions, or by using the algorithms developed for sequences. In general, the implementation of properties falls in two categories: base cases or rewrite rules. In the following subsections, each property will be explained immediately following its appearance.

### 5.4.1 Base Cases for Properties

In this subsection, properties that require separate algorithms are treated. Some properties make direct use of sequence algorithms but can not be expressed as rewrite rules, therefore these properties are also treated here.

$$\mathcal{A}^P(\mathsf{forall}\ \mathit{ident}\ \mathsf{in\ boolean}\text{: } p) \quad = \quad \textsc{Choice}(\mathcal{A}^P((p)_{\mathit{ident} \leftarrow \mathsf{true}}),\ \mathcal{A}^P((p)_{\mathit{ident} \leftarrow \mathsf{false}}))$$

The forall property repetition over a boolean is implemented by disjoining the automata for two versions of the property, one where the identifier is substituted with true and the other with false. The following rewrite is not a syntactically valid rule, although it appears logical:

$$\mathsf{forall}\ \mathit{ident}\ \mathsf{in\ boolean}\text{: } p \quad \nrightarrow \quad (p)_{\mathit{ident} \leftarrow \mathsf{true}}\ \&\&\ (p)_{\mathit{ident} \leftarrow \mathsf{false}}$$

Rewrite rules must adhere to the formal definitions of the language, and in the case above, the FL_Property conjunction operator && can not be used because forall operates on the type “Property”.

As will be explained further, property conjunction is actually implemented using automata disjunction. Since two failure matching automata representing subproperties must both hold, both automata must be triggered to find any possible failure. This explains why the Choice() algorithm was invoked in the first rule in this subsection (forall), while intuitively the forall property operator implies that each replication must hold (i.e. that their *conjunction* must hold).

$$\mathcal{A}^P(\mathsf{forall}\ \mathit{ident}\ \mathsf{in}\ \{i{:}j\}\text{: } p) \quad = \quad \textsc{ForallRange}(i,\ j,\ \mathit{ident},\ p)$$

Implementing the forall operator over a range of integers is performed using the ForallRange() algorithm presented in Algorithm 5.8. The algorithm closely resembles the implementation of the Boolean version of forall, except that multiple choice

```
1: FUNCTION: ForallRange($low$, $high$, $ident$, $p$)
2: create new automaton $\mathcal{A} = \mathcal{A}^P(\,(p)_{ident \leftarrow low}\,)$
3: for $i \leftarrow low + 1$ to $high$ do
4:     $\mathcal{A} \leftarrow$ Choice($\mathcal{A}$, $\mathcal{A}^P(\,(p)_{ident \leftarrow i}\,)$)
5: return $\mathcal{A}$
```

**Algorithm 5.8:** Forall algorithm for property $p$ that uses identifier *ident*; $i$ and $j$ are nonnegative integers.

operations are required to handle the range of values required. This is performed by the **for** loop in line 3. The loop starts at $low + 1$ because a choice operation is not required for the first instance corresponding to the case $ident \leftarrow low$ (line 2).

$$\mathcal{A}^P((f\_p)) \quad = \quad \mathcal{A}^P(f\_p)$$

Parentheses are used only for grouping and can be dropped as shown in the implementation above.

$$\begin{aligned} \mathcal{A}^P(!\,b) &\quad = \quad \textsc{FirstFail}(\mathcal{A}^B(\sim b)) \\ \mathcal{A}^P(b_1 \texttt{ <-> } b_2) &\quad = \quad \textsc{FirstFail}(\mathcal{A}^B(b_1 \texttt{ <-> } b_2)) \end{aligned}$$

Certain properties such as negation and equivalence are relegated to the Boolean layer in the simple subset. The negation and equivalency of properties, as allowed in full PSL, create properties that are not suitable for monotonically advancing time. The FirstFail() algorithm is used when Booleans and sequences are used at the property level, as explained next.

$$\begin{aligned} \mathcal{A}^P(b) &\quad = \quad \textsc{FirstFail}(\mathcal{A}^B(b)) \\ \mathcal{A}^P(s) &\quad = \quad \textsc{FirstFail}(\mathcal{A}^S(s)) \end{aligned} \qquad (5.11)$$

When Booleans and sequences are used as properties (above), their non-occurrence must be detected. The automata built by the algorithms from the previous chapter perform precisely the task of Boolean and sequence matching. When used as properties however, a Boolean or sequence's non-fulfillment indicates an error. In dynamic verification this can be interpreted as the matching of the first failure. Therefore, a separate procedure is required to transform the matching automaton into a first-failure matching automaton.

The failure algorithm implements a form of negation; however this does not correspond to classical automata negation because for run-time monitoring, it is more practical to report only the first failure for a given activation. The following example shows why negation is not appropriate in dynamic verification.

**Example 5.1:** Consider the following assertion that states that whenever $a$ is true, $b$ must be true in the next two cycles:

$$\textsf{assert always } \{a\} \mid\!=> \{b[{*}2]\};$$

The sequence $\{b[{*}2]\}$ is modeled by the automaton

$$\mathcal{A}_1 = \mathcal{A}^S(\{b[{*}2]\})$$

and describes the language

$$L(\mathcal{A}_1) = \{ \ \{b;b\} \ \}$$

Because the sequence is used as a property, its non-occurrence must be detected because it is expected to hold. The language desired by this property is modeled by the automaton

$$\mathcal{A}_2 = \textsc{FirstFail}(\mathcal{A}_1) = \mathcal{A}^P(\{b[{*}2]\})$$

whose language should intuitively correspond to

$$L(\mathcal{A}_2) = \{ \ \{\sim b\}, \{b;\sim b\} \ \}$$

As will be shown further, the FirstFail() algorithm transforms an automaton to match the first non-occurrence instead. In this example, the first failure occurs when $b$ is false in the first cycle of the right side, or when $b$ is true then followed in the next cycle by a false $b$. This represents the optimal error reporting behavior desired for run-time debugging using the assertion. The point of the example is to show that negation is not appropriate for this type of failure transformation. Below is the actual language that would be matched if negation was used (the Negate() algorithm was described in Section 4.2.4):

$$\mathcal{A}_3 = \textsc{Negate}(\mathcal{A}_1)$$
$$L(\mathcal{A}_3) = \overline{L}(\mathcal{A}_1) = \{ \ \{[{*}0]\}, \{b\}, \{\sim b\}, \{b;\sim b\}, \{\sim b;b\}, \{\sim b;\sim b\}, \{b;b;b\}, \ldots \}$$

The problem is that $L(\mathcal{A}_3)$ represents *all traces except* $\{b;b\}$, and thus reports a series of false failures, ranging from a failure in the first cycle of a successful trace, and infinite failures after three cycles or more. For these reasons the first completed

```
1: FUNCTION: FirstFail($\mathcal{A}$)   $//Q, \delta, I, F$
2: if $I \cap F \neq \emptyset$ then
3:    create new automaton $\mathcal{A}_1 = (\{q_1\}, \emptyset, \{q_1\}, \emptyset)$
4: else
5:    $\mathcal{A}_1 \leftarrow$ StrongDeterminize($\mathcal{A}$)
6:    add a new state $q_f$ to $Q_1$   $//q_f$ used as the fail state
7:    for each state $q_i \in Q_1 - \{q_f\}$ do
8:       create set of primary symbols $P = \emptyset$
9:       for each edge $(s, \sigma, d) \in \delta_1 \mid s = q_i$ do
10:         add $\sigma$'s primary symbol(s) to $P$
11:      if $P \neq \emptyset$ then
12:         for each assignment $\omega$ of primary symbols in $P$ do
13:            if $\nexists\, (s, \sigma, d) \in \delta_1 \mid s = q_i,\ (\sigma)_\omega = \mathsf{true}$ then
14:               $\sigma \leftarrow$ create or retrieve symbol in $\Sigma$ for $\omega$
15:               $\delta_1 \leftarrow \delta_1 \cup \{(q_i, \sigma, q_f)\}$
16:      else
17:         $\sigma \leftarrow$ create or retrieve existing symbol in $\Sigma$ for true
18:         $\delta_1 \leftarrow \delta_1 \cup \{(q_i, \sigma, q_f)\}$
19:   remove all edges $(s_j, \sigma_j, d_j) \in \delta_1$ for which $d_j \in F_1$
20:   $F_1 \leftarrow \{q_f\}$
21: return $\mathcal{A}_1$   $//|F_1| \leq 1$, as required for FirstFailStrong()
```

**Algorithm 5.9:** Failure matching algorithm.

failure represents much more appropriate information that should be reported by the automata. In this example $L(\mathcal{A}_2) \subset L(\mathcal{A}_3)$.

The transformation algorithm used to produce a failure matching automaton from a normal occurrence-matching automaton is shown in Algorithm 5.9. The algorithm first starts by checking for the empty match. If the argument automaton accepts the empty match (i.e. if an initial state is also a final state), then the automaton can never fail, thus the failure automaton that is returned corresponds to the null automaton (line 3).

When the input automaton does not admit the empty match, the argument automaton is determinized (line 5) and a special failure state is added (line 6). In the automata used in this work, the term deterministic does not imply completeness, thus an edge does not have to exist for each symbol in each state. To be able to detect only the first failure, the resulting automaton has to have only one active state for a given activation. In order for the automaton to be in only one state, it must be (strongly) deterministic.

The FirstFail() algorithm works by identifying the conditions where a state does

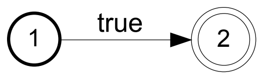


**Figure 5.6:** Automaton for $\mathcal{A}^B(\mathsf{true})$ and FIRSTFAIL($\mathcal{A}^B(\mathsf{false})$)

**Figure 5.7:** Automaton for $\mathcal{A}^B(\mathsf{false})$ and FIRSTFAIL($\mathcal{A}^B(\mathsf{true})$)

not activate a successor state. This process is repeated for each state except for the special failure state (line 7). When a state has outgoing edges, the **for** loop in line 12 and the **if** statement in line 13 add precisely the Boolean conditions where a given state does not activate a successor state. The failure conditions are incorporated onto an edge leading to the failure state, hence the failure automaton is produced. When a state has no outgoing edges, it fails for every condition therefore it directly activates the final state thru a true edge (lines 17 and 18). Line 19 performs the necessary pruning, given that former final states are no longer true final states. Consequently, former final states become unconnected and in line 13 the special final state that was added in line 6 becomes the true final state.

The failure transformation algorithm is exponential (worst case) in the number of states in the argument automaton because of the required strong determinization, although in practice the increase in number of states is manageable. The requirement for beginning in a single state for a given activation does not preclude the automaton from being retriggered at a further clock cycle for another failure matching, or even retriggered while a previous matching is taking place. In automata, multiple succeeding activations can be processed concurrently. This allows failures to be identified in a continual and dynamic manner during execution.

Figure 5.6 shows how the true automaton appears using the Boolean construction from the previous chapter. When used as a property, the failure transformation algorithm is applied to the true automaton and the null automaton in Figure 5.7 is produced. No failures can result from a matching for true, hence the null automaton results.

Alternately, Figure 5.7 also shows the false automaton (null automaton), and Figure 5.6 shows the result of applying FIRSTFAIL() to the null automaton. Since the null automaton does not match anything, it fails directly, hence the true automaton is produced by the failure algorithm.

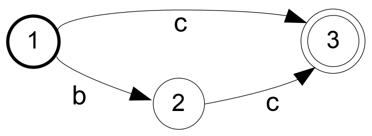


**Figure 5.8:** Automaton for $\mathcal{A}^S(\{b[*0{:}1]\,;c\})$

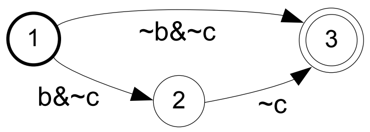


**Figure 5.9:** Automaton for FIRSTFAIL($\mathcal{A}^S(\{b[*0{:}1]\,;c\})$)

**Example 5.2:** To illustrate the effects of the FIRSTFAIL() algorithm, a sequence that matches the Boolean $c$ optionally preceded by Boolean $b$ is analyzed.

$$\{b[*0{:}1]\,;c\}$$

Figure 5.8 shows the normal matching automaton for the example sequence. The automaton's final state is activated when the trace behaves as the sequence indicates. Figure 5.9 shows how the same sequence's failures are detected, as required when the sequence is used as a property. In this case the automaton's final state is activated when the trace does not respect the behavior dictated by the sequence.

$$\mathcal{A}^P(s\,!) \quad = \quad \text{FIRSTFAILSTRONG}(\mathcal{A}^S(s)) \tag{5.12}$$

Implementing the strong sequence from Proposition 5.4 also involves constructing a failure automaton, as shown above, but with a slight modification. The FIRSTFAIL-STRONG() algorithm calls the FIRSTFAIL algorithm, and subsequently adds edges that cause the failure automaton to transition from any active state to the final state when the end of execution (EOE) signal is active (logic-1). If the automaton is processing a sequence when the EOE occurs, an error is detected and the automaton activates a final state. When a sequence completes successfully, no states are active in the failure automaton for the corresponding activation, and the EOE signal has no effect. The semantics of the EOE signal are such that the last valid cycle of execution in the finite trace is the cycle immediately before the EOE is asserted.

For example, applying the strong failure algorithm to the automaton in Figure 5.8 yields the automaton in Figure 5.10. The FIRSTFAILSTRONG() algorithm that is

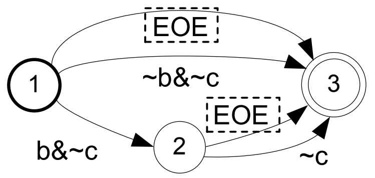


**Figure 5.10:** Automaton for FIRSTFAILSTRONG($\mathcal{A}^S$({ $b$[*0:1] ; $c$}))

```
1: FUNCTION: FirstFailStrong($\mathcal{A}$)
2: $\mathcal{A} \leftarrow$ FirstFail($\mathcal{A}$)   $//(Q, \delta, I, F)$
3: $//$here $|F| \leq 1$
4: if $|F| = 1$ (i.e. $F = \{f\}$) then
5:    $\sigma_n \leftarrow$ create or retrieve symbol in $\Sigma$ for the EOE signal
6:    for each state $q \in Q$ do
7:       $\delta \leftarrow \delta \cup \{(q,\ \sigma_n\ ,\ f)\}$
8: return $\mathcal{A}$
```

**Algorithm 5.10:** Strong failure matching algorithm.

presented in Algorithm 5.10 first builds the failure matching automaton by calling the FIRSTFAIL() algorithm (Algorithm 5.9). If the failure automaton has no final states, it is returned directly. In such cases the property can not fail, therefore the end of execution is irrelevant and there are no edges to add. If the automaton has one final state (it can have at most one, according to the FIRSTFAIL() algorithm), then lines 5 to 7 add an EOE edge from each state to the final state. This way, when the EOE signal is true *and* a state is active, the automaton indicates a failure by activating its final state.

The semantics produced by the algorithm are such that when the end of execution occurs, even if the Boolean conditions are actually satisfying the sequence, an error is signaled none the less. For example, in the property

$$\textsf{always}\ a\ \textsf{->}\ \textsf{next!}\ b$$

if $a$ occurs on the cycle before EOE is asserted, even though $b$ may be true in the $EOE$ cycle, it is too late and the automaton reports an assertion failure because the next cycle for $a$ did not occur. In this case, it was not because $b$ did not manifest itself, but rather because the execution ended before $b$'s cycle.

$$\mathcal{A}^P(f\_p\ \textsf{abort}\ b) \quad = \quad \textsc{AddLiteral}(\mathcal{A}^P(f\_p),\ \mathcal{H}(\sim b))$$

Handling the abort property also involves modifying the automaton of the property

```
1: FUNCTION: ADDLITERAL($\mathcal{A}$, $\mathcal{H}$)
2: $\sigma_l \leftarrow$ **create** or **retrieve** symbol in $\Sigma$ for $\mathcal{H}$
3: **for** each edge $(s, \sigma, d) \in \delta$ **do**
4:     $\sigma_n \leftarrow$ **create** or **retrieve** symbol in $\Sigma$ for $\sigma \wedge \sigma_l$
5:     $(s, \sigma, d) \leftarrow (s, \sigma_n, d)$
6: **return** $\mathcal{A}$
```

**Algorithm 5.11:** AddLiteral algorithm.

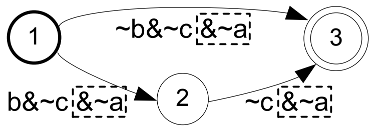


**Figure 5.11:** ADDLITERAL(FIRSTFAIL($\mathcal{A}^S$($\{b$[*0:1] ; $c\}$) , $\mathcal{H}(\sim a)$))

argument, as illustrated above. When the abort operator is encountered in the syntax tree, the automaton for the argument property $f\_p$ is built and a new primary symbol for the negated Boolean of the abort condition is created. The function ADDLITERAL() then adds a literal (a conjunct) to each edge symbol in the property automaton. The added literal corresponds to the negation of the abort's Boolean such that when the abort condition becomes true, all edges are inhibited from activating successor states, and the automaton is reset.

Since the transformation of properties to automata is also recursive in nature, for a larger property only the portion of the automaton that the abort operator was applied to will have the added literals. Furthermore, when multiple abort operators are nested in a property, each abort will contribute its own additional literal only to the portion of the whole automaton to which it applies.

The ADDLITERAL() algorithm is presented in Algorithm 5.11. The algorithm starts by creating an extended symbol for the HDL expression of the literal to be added (line 2). Then for each edge in the input automaton (line 3), a new symbol is formed with the given edge's symbol and the added literal's symbol (line 4). This new symbol replaces the old one on the given edge (line 5). An example for the abort implementation is shown in Figure 5.11. Aborting the automaton in Figure 5.9 yields the automaton in Figure 5.11. In the resulting automaton, the abort condition $a$ inhibits all edges from activating their successor states, and the automaton is reset.

$$\mathcal{A}^P(f\_p_1 \;\&\&\; f\_p_2) \quad = \quad \textsc{Choice}(\mathcal{A}^P(f\_p_1),\, \mathcal{A}^P(f\_p_2))$$

Property conjunction, not to be confused with the length matching sequence intersection, is implemented using the same algorithm that was used for sequence disjunction, namely CHOICE(). The disjunction is required because a failure by either argument property is a failure for the && property. Both argument automata are simultaneously activated by the parent node's sub-automaton in the syntax tree, and when either one reaches a final state, a failure has been detected.

The discussion on negation made earlier becomes even more relevant here, given that the implementation of property conjunction uses automata disjunction. This is reminiscent of De Morgan's law where a complemented conjunction becomes a disjunction. Negation is present in implementations of PSL for formal methods, and it should not be surprising to see it here as well. Given the run-time semantics desired for the implementation targeted by this work, the negation is not ideal.

$$\mathcal{A}^P(s \mid\!\text{->}\ f\_p) \quad = \quad \textsc{Fuse}(\mathcal{A}^S(s), \mathcal{A}^P(f\_p)) \tag{5.13}$$

Overlapped suffix implication is implemented using a sequence matching automaton directly for the antecedent sequence, which is then fused with the automaton for the consequent property, as shown above. When used in the context of properties, the fusion algorithm is the same algorithm that was devised for sequences in the previous section. The fusion algorithm introduced in Algorithm 5.5 avoids building fusion edges containing the EOE primary symbol, such that activations (antecedents) that occur at the end of execution do not cause a failure in strong properties. This was implemented by the extra condition in line 6 in the fusion algorithm.

Using fusion in properties does not create unwanted side effects, considering that the empty sequence can not cause a match in the antecedent of suffix implication. The fusion in effect ignores the empty match in both sides, which creates an automata behavior that is consistent with the formal semantics of PSL [111].

As an example, when a sequence automaton's initial state is a final state, and this sequence is used as the antecedent to overlapped suffix implication, the empty match can not cause the consequent to be checked. As was observed previously, the failure transformation algorithms can not produce automata where an initial state is also a final state (i.e. no sequence automaton, when used as a property, can accept the empty match). For these reasons, the fusion has no undesirable side effects for implementing suffix implication.

**Example 5.3:** Fusion allows the proper processing of the following assertions:

assert always $\{a\}$ |-> { $\{b\}$ && $\{b;b\}$ };

$$\textsf{assert always } \{b[\text{*}0{:}1]\} \mid\!\!\text{->}\ p;$$

In the first line, the length-matching intersection results in a property automaton for the consequent that is identical to Figure 5.6 (the true automaton). The assertion thus fails whenever $a$ is observed. In the second line, $b$ must be asserted *once* in order for the consequent to be checked.

Items in Proposition 5.4 not treated thus far do not need to be explicitly handled in the checker generator kernel. If such properties can be expressed using other properties or sequences, they are rewritten when they are encountered during checker generation. In such cases, these operators are transparent to the checker generator kernel. The properties that can be implemented by rewrite rules all have the form $\mathcal{A}^P(x) = \mathcal{A}^P(y)$, and are implemented with a rewrite rule of the form $x \rightarrow y$, similarly to what was done for sequences.

Using the sugaring definitions for properties from Appendix B in the PSL specification [111] as rewrite rules is generally not feasible because of the restrictions imposed by the simple subset. For this purpose, a set of rewrite rules is introduced that is suitable for the simple subset of PSL, within the context of dynamic verification. The rules are not intended to extend upward to full PSL.

The following sugaring definition shows an example of why such definitions can generally not be used as rewrite rules:

$$\textsf{always}\ p \ \stackrel{\text{def}}{=}\ \neg\ \textsf{eventually!}\ \neg p \qquad (\textsf{G}\, p \ \stackrel{\text{def}}{=}\ \neg\, \textsf{F} \neg p \ \ [111])$$

The above definition for always can not be used in the simple subset because negating a full property is not defined. Moreover, the argument of the eventually! operator is restricted to a sequence, and sequences can not be negated. Rewrite rules that are compatible with the simple subset must be developed for the checker generator.

In some cases, the easiest way to handle an operator is by rewriting it using a more complex operator. Since the more complex operator has to be handled, particular code for the simpler cases is avoided. For example, rewriting next_a using next_event_a may appear overly complex; however, since the next_event_a operator already exists and must be supported, it subsumes all simpler forms of this family of operators. The rewrite rules for properties are categorized in three groups and are treated in the following three subsections. Although a few rewrite rules may appear intuitive, they are nonetheless included for completeness. Each will be explained immediately following its appearance.

### 5.4.2 Rewrite Rules Based on Suffix Implication

The rewrite rules in this subsection all have the common characteristic that the right side makes use of the overlapped suffix implication operator defined in properties. Unless specified, the term *suffix implication* is meant as a short form for overlapped suffix implication (as opposed to non-overlapped suffix implication). As was the case in sequences, proper care must be taken to ensure that the rewrite rules are terminating (Definition 2.13), and that no infinite loop is possible.

$$b \mid\mid f\_p \quad \rightarrow \quad \{{\sim}b\} \mid\text{->} f\_p$$

In the simple subset, one of the properties used in disjunction must be Boolean. The rewrite rule is based on the fact that if the Boolean expression is not true, then the property must be true; otherwise the property is automatically true. For simplicity in the presentation of the || operator in Proposition 5.4 (and in Definition 2.10), the Boolean expression is shown as the left argument; an equally acceptable disjunction could have the form $f\_p \mid\mid b$.

$$b \text{ ->} f\_p \quad \rightarrow \quad \{b\} \mid\text{->} f\_p$$

Since a Boolean in curly brackets is a valid sequence, the property implication above can be rewritten using a suffix implication.

$$s \mid\text{=>} f\_p \quad \rightarrow \quad \{s \,;\, \mathsf{true}\} \mid\text{->} f\_p$$

The rewrite rule for non-overlapped suffix implication above follows from its sugaring definition in Appendix B in the PSL specification [111]. The simple subset does not affect this definition, therefore it can be used directly as a rewrite rule. The approach consists of concatenating an extra true cycle at the end of the antecedent sequence.

$$\mathsf{always}\ f\_p \quad \rightarrow \quad \{[+]\} \mid\text{->} f\_p \tag{5.14}$$

As explained in Chapter 2, suffix implication has a sequence as the antecedent, and a property as a consequent. When a property must always be true, it can be expressed as the consequent of a suffix implication with a perpetually active antecedent ([+] is sugaring for true[+]).

$$\mathsf{never}\ s \quad \rightarrow \quad \{[+] : s\} \mid\text{->} \mathsf{false}$$

When a sequence must never occur, a property that fails instantly is triggered upon detection of the sequence. Since the sequence must never occur, it is continually activated by the fusion with [+]. The overlapped suffix implication does not have a clock cycle delay between antecedent and consequent thus the rewrites for never and always offer the correct timing. The never operator also accepts a Boolean as its argument but for simplicity was not treated. In that case, the following rewrite rule can be used to express the Boolean as a simple sequence.

$$\mathsf{never}\ b \quad \rightarrow \quad \mathsf{never}\ \{b\}$$

It should be noted that the implementation of never described above does *not* respect the semantics defined in the PSL specification [111]. As stated in an issue to be addressed in the next revision of PSL [113], if $r$ is a SERE, "never $\{r\}$" behaves in a counter-intuitive manner when a finite trace terminates while $r$ is being matched. Intuitively, it is instead the strong form "never $\{r\}!$" that should be specified. The rewrite rule above has been proven to align with the strong never, and thus offers the correct semantics. The proof was conducted by Morin-Allory using the PVS theorem prover [132]. To summarize the problem, the strong never is not allowed in the simple subset of PSL, yet it is its semantics which are the most appropriate for the run-time behavior of the never operator.

$$f\_p\ \mathsf{until}\ b \quad \rightarrow \quad \{(\sim b)[+]\}\ |\text{->}\ f\_p$$

The until operator states that the property $f\_p$ must be true on each cycle, up-to, but not including, the cycle where the Boolean $b$ is true. In the rewrite rule above, the implication has the effect of enforcing the property $f\_p$ for each cycle of consecutive $\sim b$s. In the run-time semantics used in this work for the until operator, the property is allowed to fail for multiple start times when the Boolean $b$ is continuously false.

$$f\_p\ \mathsf{until\_}\ b \quad \rightarrow \quad \{\{(\sim b)[+]\}\,|\,\{b[\text{->}]\}\}\ |\text{->}\ f\_p$$

The implementation of the overlapped operator until_ is similar to the non-overlapped until operator with the addition of a condition for enforcing the checking of the property $f\_p$, namely that it must also hold for the cycle where the Boolean $b$ is true.

$$\mathsf{next\_event\_a}(b)[k{:}l](f\_p) \quad \rightarrow \quad \{b[\text{->}k{:}l]\}\ |\text{->}\ f\_p$$

The next_event_a property above states that all occurrences of the next event within the specified range must see the property be true. This can be modeled using a goto repetition with a range as an antecedent to the property. The antecedent triggers the monitoring of the property $f\_p$ each time $b$ occurs within the specified range. Once more the suffix implication operator was used as a key component in a rewrite rule.

## 5.4.3 Rewrite Rules Based on Sequences as Properties

Some properties from Proposition 5.4 can be implemented by expressing their semantics using sequences. Since the property is rewritten to a sequence, the sequence takes the place of the property; the sequence is thus interpreted as a property, using the implementation in (5.11) that makes use of the FirstFail() algorithm.

$$b_1 \ \mathsf{before} \ b_2 \quad \rightarrow \quad \{(\sim b_1 \& \sim b_2)[*] \ ; \ (b_1 \& \sim b_2)\}$$

The before property above can be modeled by a sequence. When $b_1$ should happen strictly before $b_2$, it can be expected that $b_1$ will occur simultaneously with $\sim b_2$ (right side of the concatenation). However, this does not need to happen immediately, therefore the condition can be preceded with an arbitrary number of cycles where both Booleans are false (left side of the concatenation). All other outcomes indicate a violation the expected behavior, and since this sequence is used as a property, the FirstFail() algorithm resulting from (5.11) will detect these violations.

$$b_1 \ \mathsf{before\_} \ b_2 \quad \rightarrow \quad \{(\sim b_1 \& \sim b_2)[*] \ ; \ b_1\}$$

The overlapped operator before_ states that $b_1$ must be asserted before or simultaneously with $b_2$. This rewrite is very similar to the previous rewrite, with the exception that the constraint on $b_2$ is relaxed on the right side of the concatenation. This indicates that when $b_1$ is matched, $b_2$ could have been true also, and the overlapped case is therefore allowed, as required by the property's semantics.

$$\begin{aligned} b_1 \ \mathsf{before!} \ b_2 \quad &\rightarrow \quad \{(\sim b_1 \& \sim b_2)[*] \ ; \ (b_1 \& \sim b_2)\}! \\ b_1 \ \mathsf{before!\_} \ b_2 \quad &\rightarrow \quad \{(\sim b_1 \& \sim b_2)[*] \ ; \ b_1\}! \end{aligned}$$

The rewrite rules for the strong versions of the before properties are very similar to the rules for the weak versions presented previously, with the exception that strong sequences are used, thereby indicating that they must complete before the end of execution. Strong sequences are implemented in (5.12).

$$\mathsf{next\_event\_e}(b_1)[k{:}l](b_2) \quad\rightarrow\quad \{b_1[\text{->}k{:}l] : b_2\}$$

The next_event_e property states that the Boolean $b_2$ should be asserted at least once in the specified range of next events of the Boolean $b_1$. This behavior is modeled using a goto repetition that is fused with the $b_2$ consequent. Since the sequence is used as a property, once the $b_2$ consequent is observed in the proper range, the sequence has completed successfully and will not indicate a failure. All other conditions will be reported as failures by the implementation of sequences at the property level in (5.11).

$$\mathsf{next\_event\_e!}(b_1)[k{:}l](b_2) \quad\rightarrow\quad \{b_1[\text{->}k{:}l] : b_2\}!$$

The strong version of the next_event_e property is implemented similarly to its weak counterpart, with the exception that a strong sequence is instead used.

$$\mathsf{eventually!}\ s \quad\rightarrow\quad \{[+] : s\}! \tag{5.15}$$

Rewriting the eventually! operator is done by enforcing that the sequence $s$ must complete before the end of execution. The sequence may start at any cycle after the eventually! property is activated, hence the fusion with [+]. For non degenerate sequences $s$, the semantics of the rewrite are such that if the sequence is not observed when the end of execution occurs, the property fails.

For degenerate sequences $s$, the failure can be reported sooner than at the end of execution. If an empty or null sequence is used for $s$, the fusion with [+] returns a null automaton (no final state; Figure 5.2). Subsequently, when the FirstFailStrong() algorithm is applied in (5.12), the true automaton is returned. The true automaton was shown in Figure 5.6, and corresponds to a simple two state automaton with the Boolean true as its edge symbol. When a property corresponds to the true automaton (the true property), it fails on the cycles were it is activated. Moreover, if the true property is the top-level argument of an assert directive, the property fails on the first cycle after the reset is released.

The eventually! operator also accepts a Boolean as its argument but for simplicity was not treated. In that case, the following rewrite rule can be used to express the Boolean as a sequence.

$$\mathsf{eventually!}\ b \quad\rightarrow\quad \mathsf{eventually!}\ \{b\}$$

The following example illustrates the semantics of the eventually! rewrite.

**Example 5.4:** The assertions below are used to illustrate the semantics of the eventually! rewrite when degenerate sequences are used.

assert eventually! {[*0]};

assert eventually! {{true} && {true[*2]}};

assert always ($a$ -> eventually! {[*0]});

assert always ($a$ -> eventually! {{true} && {true[*2]}});

The first two assertions fail in the cycle immediately after reset is released, and the end of execution signal (EOE) does not have to be monitored. In the last two assertions above, the assertions fail in any cycle where $a$ is true, because it is known that the degenerate sequences will never be matched (the empty sequence is not a valid match at the property level). The EOE signal is also not required in those cases. To put the above examples in context, consider the following assertion.

assert always ($a$ -> eventually! {$b$[*0:1]});

This assertion can only fail when the end of execution occurs, provided $a$ has occurred at least once and that the last occurrence of $a$ has not seen its future occurrence of $b$.

If it is desired that all the failures for the eventually! operator be only reported at the end of execution, whether degenerate or normal sequences are used, an alternate but also valid solution can be devised using the strategy below.

$$\mathcal{A}^P(\textsf{eventually! } s) \quad \rightarrow \quad \begin{cases} \mathcal{A}^P(\{[+] : s\}!) & \text{if } s \text{ is non-degenerate} \\ \mathcal{A}^P(\textsf{never } \{EOE\}) & \text{otherwise} \end{cases} \qquad (5.16)$$

The first case corresponds to the rewrite introduced previously. The second case represents a compact way of building an automaton that triggers every time the end of execution signal ($EOE$) is activated.

### 5.4.4 Rewrite Rules Based on Property Variations

The rewrite rules in this subsection are based on variations of other properties.

$$f\_p \textsf{ until! } b \quad \rightarrow \quad (f\_p \textsf{ until } b) \textsf{ \&\& } (\{b[\text{->}]\}!)$$

$$f\_p \text{ until!\_ } b \quad \rightarrow \quad (f\_p \text{ until\_ } b) \text{ \&\& } (\{b[\text{->}]\}!)$$

The strong versions of the until properties are created by using the weak versions and adding a temporal obligation for the releasing condition to occur, namely $b$. This can be modeled by the strong single-goto of the Boolean condition $b$. If the end of execution occurs before the releasing condition $b$ occurs, the assertion will trigger even though the weak until may have always held.

$$\begin{aligned}
\text{next } f\_p &\quad\rightarrow\quad \text{next}[1](f\_p)\\
\text{next! } f\_p &\quad\rightarrow\quad \text{next!}[1](f\_p)\\
\text{next\_event!}(b)(f\_p) &\quad\rightarrow\quad \text{next\_event!}(b)[1](f\_p)\\
\text{next\_event}(b)(f\_p) &\quad\rightarrow\quad \text{next\_event}(b)[1](f\_p)
\end{aligned}$$

The rewrites above implement the basic next and next_event operators by rewriting them to slightly more complex forms. These rules are based on the fact that when no count is specified, a count of 1 is implicit. Since the right sides of these rules are not terminal, they are subsequently rewritten using other rules, until no more rewrites apply and either sequences, Boolean expressions or base cases of properties are reached.

$$\begin{aligned}
\text{next}[i](f\_p) &\quad\rightarrow\quad \text{next\_event(true)}[i+1](f\_p)\\
\text{next!}[i](f\_p) &\quad\rightarrow\quad \text{next\_event!(true)}[i+1](f\_p)\\
\text{next\_a}[i{:}j](f\_p) &\quad\rightarrow\quad \text{next\_event\_a(true)}[i+1:j+1](f\_p)\\
\text{next\_a!}[i{:}j](f\_p) &\quad\rightarrow\quad \text{next\_event\_a!(true)}[i+1:j+1](f\_p)\\
\text{next\_e}[i{:}j](b) &\quad\rightarrow\quad \text{next\_event\_e(true)}[i+1:j+1](b)\\
\text{next\_e!}[i{:}j](b) &\quad\rightarrow\quad \text{next\_event\_e!(true)}[i+1:j+1](b)
\end{aligned}$$

The family of six rewrite rules above is all based on the fact that next_event is a more general case of next. The "+1" adjustment is required to handle the mapping to the Boolean true. When converting a next property to a next_event property, there is a slight nuance as to what constitutes the next occurrence of a condition. The next occurrence of a Boolean expression can be in the current cycle, whereas the plain next implicitly refers to the next cycle.

Another reasoning shows the consistency between the operators: $\text{next}[0](f\_p)$ could not be modeled without the increment because $\text{next\_event}(b)[k](f\_p)$ requires a positive count for $k$. The operator next[0]($f\_p$) is also equivalent to $f\_p$.

$$\begin{aligned} \mathsf{next\_event}(b)[k](f\_p) &\quad\rightarrow\quad \mathsf{next\_event\_a}(b)[k{:}k](f\_p) \\ \mathsf{next\_event!}(b)[k](f\_p) &\quad\rightarrow\quad \mathsf{next\_event\_a!}(b)[k{:}k](f\_p) \end{aligned}$$

The strategy behind the above rewrite rules is to utilize the next_event_a form, with identical upper and lower bounds for the range.

$$\mathsf{next\_event\_a!}(b)[k{:}l](f\_p) \quad\rightarrow\quad \mathsf{next\_event\_a}(b)[k{:}l](f\_p)\ \&\&\ \{b[\text{->}l]\}!$$

The rewrite above implements the next_event_a! property. Similarly to the rewrite for the until! property, it is rewritten using the weak version, to which a necessary completion criterion is conjoined. The addition of the strong goto sequence with the $l$ bound indicates that all $l$ occurrences of the $b$ event must occur before execution terminates.

## 5.5 Implementation of Verification Directives

The automaton implementation of the verification directives from Definition 2.11 is presented below in Proposition 5.5.

**Proposition 5.5:** If $s$ is a Sequence, $p$ is a Property and $v$ is a verification directive, then the automaton implementation of *verification directives*, denoted $\mathcal{A}^V(v)$, is performed as follows:
$\mathcal{A}^V(v)$ :

- $\mathcal{A}^V(\mathsf{assert}\ p;) \quad = \quad \mathcal{A}^P(p)$
- $\mathcal{A}^V(\mathsf{cover}\ s;) \quad = \quad \mathcal{A}^V(\mathsf{assert\ eventually!}\ s;)$

The assert directive is implemented by directly returning the automaton created for its argument property. The property identifies all faulty traces, and nothing more needs to be done for implementing the assert directive.

$$\mathsf{cover}\ s \quad\rightarrow\quad \mathsf{assert\ eventually!}\ s$$

The cover directive provides a way to ensure that a given sequence has eventually manifested itself during the verification. It is therefore natural to rewrite the directive using the eventually! property.

Although the default clock declaration is not a PSL directive as such, clock declarations and verification directives are the items that are used in verification units in the PSL presented in this work. Verification items and default clock declarations are the minimum statements that must be specified by the user to get the verification process up-and-running. An example vunit is shown in Appendix A. To conclude the chapter, a summary example is presented to show the multiple steps required to build an automaton for an assertion.

**Example 5.5:** The following assertion states that whenever $\sim a$ is followed by $a$, then in the cycle where $a$ was true the sequence $\{b[*0{:}1];c\}$ must occur. This sequence states that $c$ must occur, possibly preceded by $b$. For simplicity the example assertion uses simple Booleans; however, complex Boolean expressions over real circuit signals can also be used.

assert always $\{\sim a\ ;\ a\}$ |-> $\{b[*0{:}1]\ ;\ c\}$;

In dynamic verification, the assertion output will trigger in every cycle where a non-compliance to the assertion is observed in the input signals. Figure 5.12 shows six steps for creating the automaton for the assertion above. Each part of the figure is explained below.

a) The terminal Boolean automata are built for the Booleans used in the assertion according to the BaseCase() algorithm in Algorithm 5.1.

b) The repetition range and concatenation algorithms are used with the Boolean automata in part a) to construct the automata for the two sequences used in the suffix implication. Since these sequences are top-level sequences, minimization was applied.

c) The right side sequence is used as a property therefore the first failure must instead be matched. This transformation is provided by the FirstFail() algorithm according to (5.11).

d) The suffix implication in the assertion is implemented using the Fuse algorithm, as shown in (5.13).

e) The always operator is rewritten according to (5.14), and is implemented as a suffix implication using the sequence {[+]} as the left side, and the property that is the argument of always as the right side. The automaton for {[+]} is shown in the left side of part e), and requires a few steps to build (not shown). The suffix implication that is part of the always rewrite is implemented with a fusion.

f) The fusion mentioned at the end of part e) is performed, and after minimization the final resulting automaton for the example assertion is produced.

Converting the automaton to RTL can subsequently be performed as explained in Section 4.3. The checker is then ready to be used for in-circuit assertion monitoring.

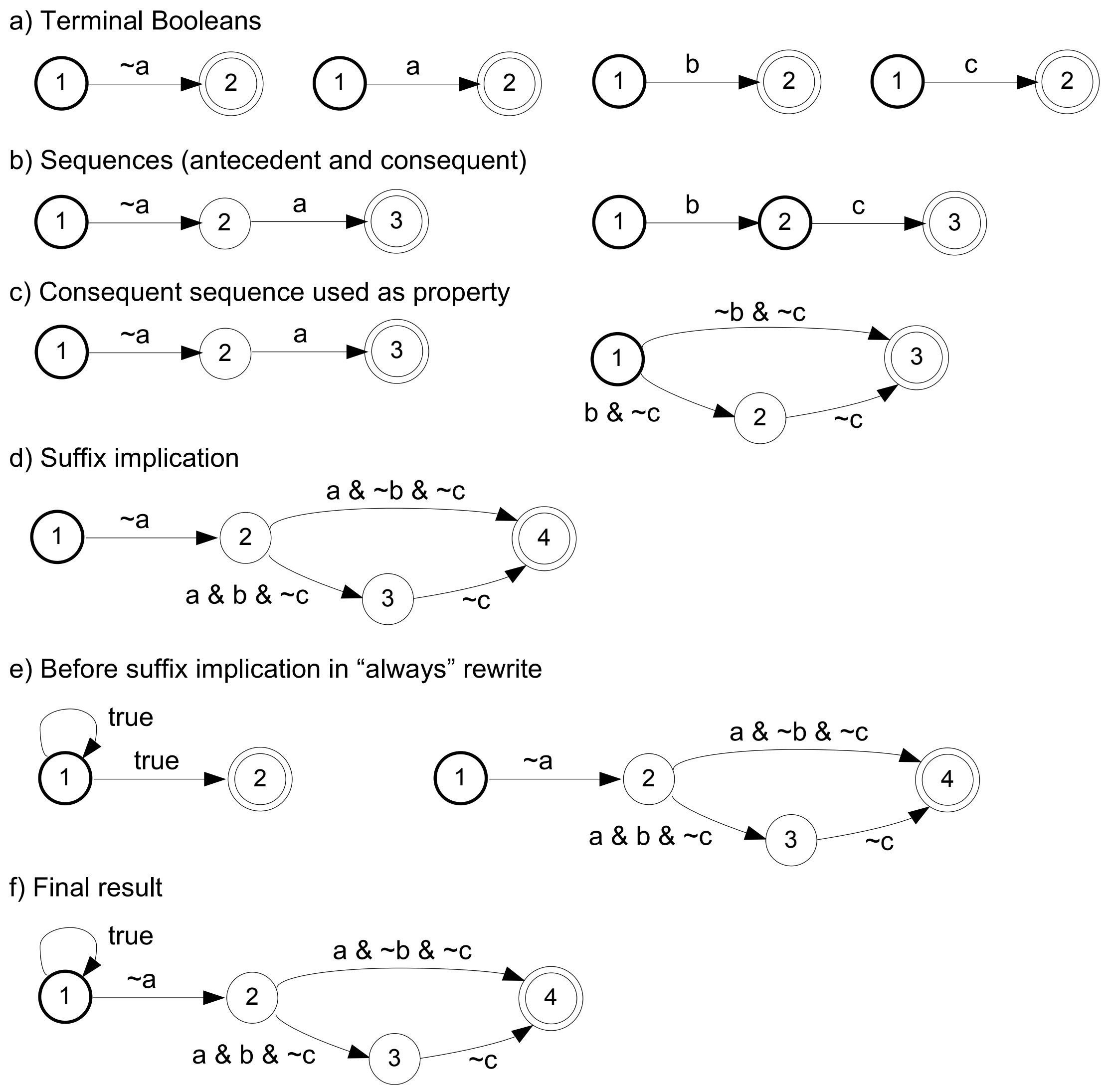


**Figure 5.12:** Complete checker for example: assert always $\{\sim a\,;a\}$ |-> $\{b[*0{:}1]\,;c\}$;. Steps a) to f) show the bottom-up construction starting with terminal Booleans and ending with the resulting automaton.

# Chapter 6

# Enhanced Features and Uses of Checkers

## 6.1 Introduction and Overview

The topics contained in Chapters 4 and 5 show how a circuit-level checker can be generated from a PSL assertion for use in the ABV paradigm. In this chapter, a set of enhancements to assertion checkers is introduced. The first enhancement is a more resource-efficient implementation of the eventually! operator, and is presented in Section 6.3.

Section 6.4 introduces a multitude of debug enhancements for checkers, which all share a common goal of improving the debugging process through improved observability in the checkers. Finding a failure is only the first step in the bug fixing process, and the exact cause of the failure must be determined before performing subsequent design changes. Specifically, Section 6.4 proposes a set of additions and modifications to the checkers to help identify the causes of errors more easily and to help decrease the time spend debugging a circuit. Assertions themselves are only a "foot in the door" for the debugging process, as engineers still need to invest time to explore the reasons why an assertion failed. Assertions help to find the existence of errors, and with more sophisticated techniques, assertions should also help to identify the true causes of the errors, since the implicit goal is often to fix any errors that are found. For example, knowing that a bus grant was not given in the proper time frame by an arbiter doesn't directly reveal the actual functional reason for that error.

In order for some of these checker enhancements to be possible, a modified recursive compilation strategy must be introduced. This is done in the next section, where the necessary modifications to perform automata splitting (or modularizing) are introduced. Automata splitting is a technique whereby a subset of an automaton is kept isolated from the remainder of an assertion's automaton. In order for the isolated automaton to be properly triggered, it is given a precondition automaton during compilation, which represents its activations. The use of precondition automata is a prerequisite for automata splitting. Automata splitting is required for the assertion threading and the activity monitoring enhancements, and also for the more efficient form of eventually!. The efficient eventually! implementation described in this chapter is used by default in the checker generator, as opposed to the rewrite rule developed in the previous chapter.

The checker generator also has an option to force the use of a single automaton to represent an assertion's checker. In such cases, the efficient form of eventually! is not used, and the rewrite is applied. Automata threading and activity monitors are also overridden, when a single automaton checker is desired.

The use of assertion checkers beyond verification purposes is explored in Section 6.5. There, it is shown that assertion checkers also play an important role in post-fabrication silicon debug and on-line monitoring, and that using assertions and a checker generator can even be used for performing certain types of circuit design. Although these themes are not fully checker enhancements as such, they can nonetheless be considered as enhanced applications of checkers.

## 6.2 Recursive Compilation Strategies

The implementation of PSL properties in Section 5.4 was presented in functional form, where functions are called with automata as arguments, and the functions each return an automaton. The functional form also applies to the base cases in Section 5.4.1, whereas the rewrite rules revert to the bases cases after one or more applications. Functions such as CHOICE(), FUSE(), ADDLITERAL() were used to implement property operators when rewrite rules were not used. This compilation strategy yields the simplest and most direct architecture in the checker generator. However, for many of the enhancements presented in this chapter, this recursive strategy is not appropriate.

In certain cases, a node in the recursive compilation may require that its sub-automaton be instantiated as a module, and not be merged with it's parent's automaton. The most evocative example of this is when monitoring sequence activity is to be performed. In order for a sequence's activity to be monitored properly, its

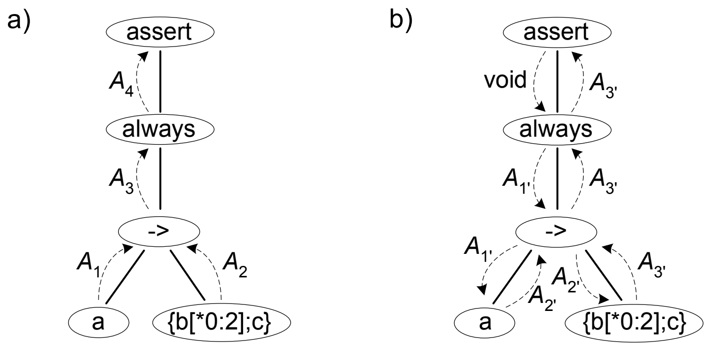


**Figure 6.1:** Example of recursive compilation strategies: a) without precondition automata, b) with precondition automata.

automaton must retain its structural integrity, and it must not be merged with other automata. In these cases, the parent node has to supply the child with a *precondition* automaton, such that if the child node decides that its automaton needs to be a module, the child has everything at its disposal to instantiate its activation automaton. This recursive compilation strategy is also required for implementing the more efficient form of eventually! in Section 6.3, as well as the activity monitors and assertion threading in Section 6.4.2 and Section 6.4.5.

The following assertion is used as a running example to illustrate the two recursive compilation strategies.

$$\textsf{assert always } a \textsf{ -> } \{b[*0{:}2];c\}; \qquad (6.1)$$

When the assertion is compiled using the algorithms presented in the previous chapter, the compilation graph shown in Figure 6.1 a) applies. This graph is not explicitly stored in memory; it only represents the recursive function calls that are made, as dictated by the syntax structure of the assertion (syntax tree). The compilation strategy is characterized by the fact that a parent sends no information to its child, which is itself responsible for building an automaton for its sub-property, and nothing more. The child is not aware of the upper level context and it is the parent that is responsible for building its own property automaton. This strategy is said to be *without precondition automata.*

In the recursive strategy *with precondition automata*, as shown in Figure 6.1 b), a parent node must send each child an automaton representing the child's activations. If the child does not require its sub-property automaton to be isolated, this automaton is built and the precondition automaton is fused with it (by the child) so that it may be activated properly. Preconditions may be void, in which case the fusion is not required. A *void precondition* can only be produced by the assert operator, and is equivalent to indicating that the child property must only hold starting in the first cycle of execution.

Analyzing more deeply the example without precondition automata in Figure 6.1 a), it can be observed that the terminal property elements are the antecedent Boolean and the consequent sequence for the implication. Each of these elements is assembled in an automaton form using the techniques for Booleans and sequences from the previous chapter. The $A_1$ and $A_2$ automata are built as follows:

$$A_1 = \mathcal{A}^B(a)$$
$$A_2 = \mathcal{A}^S(\{b[*0{:}2];c\})$$

These automata are returned to the implication node, where the resulting automaton for the implication is built. Using the techniques from the previous chapter, it follows that:

$$A_3 = \textsc{Fuse}(A_1, \textsc{FirstFail}(A_2))$$

This automaton is returned to the always node, where it is perpetually activated as a result of the always rewrite:

$$A_4 = \textsc{Fuse}(\mathcal{A}^S(\{[+]\}), A_3)$$

The resulting automaton is $A_4$, and is left intact by the assert operator.

The same automaton can also be built using the precondition-automata strategy. In Figure 6.1 b), the automata indices are not related to those in Figure 6.1 a), and are analyzed in detail next. The assert operator implicitly represents an activation for the first cycle only, and does not require a precondition automaton for its child property (hence the void automaton). The always node prepares its precondition automaton as follows, to indicate that its child should be perpetually checked:

$$A_{1'} = \mathcal{A}^S(\{[+]\})$$

Once received by the implication node (->), the precondition automaton $A_{1'}$ is redirected to the antecedent node $a$. The result automaton from the antecedent child is formed as follows:

$$A_{2'} = \textsc{Fuse}(A_{1'}, \mathcal{A}^B(a))$$

The result automaton $A_{2'}$ is returned to the implication node, where it is redirected to the other child to serve as its precondition automaton. The right-side child in the syntax tree builds the sequence automaton $A_{3'}$ as follows:

$$A_{3'} = \textsc{Fuse}(A_{2'}, \textsc{FirstFail}(\mathcal{A}^S(\{b[*0{:}2];c\})))$$

An important subtlety is that the child sequence has to be made aware that it is being used as a property, and that its failure matching must be performed. In the approach without precondition automata, the $\textsc{FirstFail}()$ was not performed by the child, but rather by the parent (i.e. the implication). Getting back to the example in Figure 6.1 b), now that the consequent node's automaton is ready, it is returned back to the implication node, that directly returns it to the always node. In this strategy, the child has done the implication's fusion, and nothing else needs to be done at the -> node. The always node also does not have to do anything, since its continual activation was modeled in the precondition automaton that was sent to its child. Thus $A_{3'}$ is returned as the resulting automaton.

Substituting $A_1$ and $A_2$ into $A_3$, and then substituting the result into $A_4$ yields the following computation for the assertion's automaton (without using precondition automata):

$$\textsc{Fuse}(\mathcal{A}^S(\{[+]\}), \textsc{Fuse}(\mathcal{A}^B(a), \textsc{FirstFail}(\mathcal{A}^S(\{b[*0{:}2];c\})))) \tag{6.2}$$

Performing the same type of substitutions for $A_{3'}$ yields the expression (for the precondition automata strategy):

$$\textsc{Fuse}(\textsc{Fuse}(\mathcal{A}^S(\{[+]\}), \mathcal{A}^B(a)), \textsc{FirstFail}(\mathcal{A}^S(\{b[*0{:}2];c\})))$$

Because the fusion operator is associative, the expression above is functionally equivalent to the expression in (6.2). This shows that both recursive compilation strategies are equivalent in this example. Although both strategies have the same effect, the method using precondition automata is more flexible and allows automata to be modularized, as required by some of the enhancements introduced in this chapter.

Precondition automata are used only at the property level, and SEREs and Booleans are compiled as implied by the functional notation in the previous chapter. Only a few other property operators from Proposition 5.4 require special treatment for the precondition automaton strategy. For unary operators, a child node incorporates the precondition automaton to its own automaton using automata fusion. The precondition in effect represents the *activations* of the child node's expression. In the fusion, the precondition automaton is the left argument, and the right argument is the automaton for the sub-property rooted at the child node.

For the binary operators corresponding to overlapped suffix implication and property implication, the strategy consists in: 1) sending a precondition automaton to the antecedent; 2) using the result automaton from the antecedent as the precondition automaton for the consequent; 3) using the result automaton from the consequent as the result automaton for the implication itself. In the compilation strategy with precondition automata, the parent actually tells a child node whether or not it is in normal matching mode (ex: antecedent) or in failure mode (ex: consequent). This nuance was observed when comparing both strategies in the example for the assertion in (6.1), where the FirstFail() for the consequent sequence is applied by the *implication node* when preconditions are not used, and the FirstFail() is applied in the *consequent sequence node* when preconditions are used.

For the property conjunction operator &&, the node sends its precondition automaton simultaneously to both arguments, and applies the Choice() algorithm to the automata returned by its children nodes. The abort operator still uses the AddLiteral() algorithm to reset a sub-automaton, except that the abort is implemented by the children nodes before the fusion with the precondition automaton. The abort's Boolean must be kept in a global string, where nested aborts are added as disjunctions to the global abort string. The abort string starts out as the null string, and grows/shrinks as the recursive traversal in the syntax tree progresses.

To summarize, the essential points of the two recursive compilation strategies are stated below. In both cases, the compilation of a PSL property involves recursively scanning the syntax tree of the PSL expression.

1. *Without precondition automata*: Each node returns an automaton describing the behavior of the expression rooted at that node. The parent node then builds its own automaton from its children automaton(s), using a variety of transformations and operations.
2. *With precondition automata*: Each node sends a precondition in automaton form to a child, whereby the *child* node is responsible for building the sub-automaton (with its activations) and returning it to the parent. If other child nodes exist, the parent forms other precondition automata, possibly using the automata returned by previous child nodes. When finished, the parent returns an automaton formed from the children automata (directly or with modifications) to its parent.

The first recursive strategy is implied by the functional form used in the previous chapter, and was initially used in the checker generator. When the debug enhancements were added, the precondition method was implemented, and is currently the technique used in the checker generator. The precondition/result framework has its roots in the first version of the checker generator [29], which implemented properties and sequences modularly using precondition and result signals.

## 6.3 A Special Case for eventually!

In the previous chapter it was stated that rewriting the eventually! operator can be performed with the rewrite rule in (5.16), recalled below:

$$\mathcal{A}^P(\textsf{eventually! } s) \quad \rightarrow \quad \begin{cases} \mathcal{A}^P(\{[+]:s\}!) & \text{if } s \text{ is non-degenerate} \\ \mathcal{A}^P(\textsf{never } \{EOE\}) & \text{otherwise} \end{cases} \tag{6.3}$$

This strategy has the advantage of preserving the full automaton approach; however, given that the sequence in the right-hand side of the non-degenerate case is used as a property, the FirstFailStrong() algorithm has to be applied to the sequence. That algorithm requires a proper determinization, and thus does not represent the most efficient solution. This section details the use of a more efficient procedure for implementing the eventually! property, in cases where automata splitting is allowed and the use of separate logic and wire declarations are permitted. An efficient implementation of eventually! is also important for the cover directive which is rewritten to the eventually! operator in dynamic verification. Although automata optimizations can no longer cross the boundaries of split automata, the improvements brought upon by the split approach outweigh this disadvantage.

In the split automata approach, implementing the "eventually! $s$" property is done with a normal occurrence-matching automaton. After the automaton for the sequence $s$ is built, its initial states are made non-final. At this point, if the sequence automaton has no final states, the sequence can not eventually occur, and the failure must be signaled at the end of execution. In this case the automaton corresponding to $\mathcal{A}^P(\textsf{never } \{EOE\})$ is returned to the parent node in the syntax tree of the assertion, similarly to the degenerate case of the rewrite rule in (6.3).

If the sequence automaton is not degenerate, then a more complex algorithm is applied. First, the automaton is weakly determinized such that it has only one initial state. Then, any incoming edges to the initial state are removed, and outgoing edges from the final states are removed. Incoming edges to the initial state are redundant since the state will remain active until the sequence is matched. Outgoing edges of the final states can be safely removed since the first match of the sequence is sufficient to satisfy the eventually!. The automaton must be implemented as a module, for which a result signal is declared. This result signal is then used, complemented, as the symbol of a self-loop on the initial state. This has the effect of keeping the initial state active until at least one occurrence of the sequence has manifested itself.

The actual result signal of the eventually! operator corresponds to the output of the initial state's flip-flop. In this manner, no extra states (hence flip-flops) are used. The actual result signal is implemented in automaton form before being returned to eventually!'s parent node. This consists in preparing a two-state automaton where the second state is a final state, the initial state has a true self-loop, and an edge from the initial state to the second state carries a symbol corresponding to the result signal.

When a precondition automaton is passed to the eventually! node in the recursive compilation, the precondition automaton is implemented as a module, for which a result signal is declared. This signal constitutes the precondition signal for the eventually! automaton. Each time the precondition is asserted, the conditional mode automaton for eventually! is flushed, with the start and final state's edges modified as described previously. Automaton *flushing* consists in deactivating the edges for all states except the initial state. This is accomplished by AND-ing a negated literal of the precondition signal to each outgoing edge symbol of each non-initial state. In this manner, each new precondition guarantees a new complete obligation. The precondition automaton used in this technique implies that the recursive mechanism *with precondition automata* must be used.

**Example 6.1:** Figure 6.2 shows an example of the efficient eventually! strategy for the following property:

$$\text{always } (a \text{ -> eventually! } \{b;c;d\})$$

The property is actually implemented as two automata, and the automaton at the top right in the figure is returned by the always node in the compilation tree. Since the always property is the argument of the assert directive, the returned automaton is directly implemented in RTL. The grey state also serves as the memory state, which is deactivated once the obligation is fulfilled (once the sequence occurred). Automaton flushing is also visible with the added "$\sim s2$" literals. If the always property was part of a more complex assertion, the returned automaton would be used by the parent property to continue to build the complete automaton for the assertion.

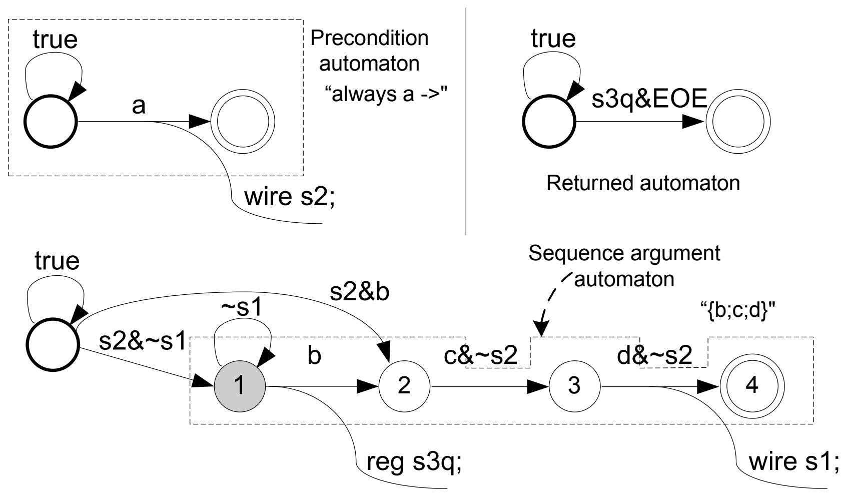


**Figure 6.2:** Efficient implementation of the eventually! assertion in Example 6.1.

Automata splitting and the addition of separate logic gates could also be used for performing efficient automata negation by avoiding the determinization step. In hardware, an NFA could be easily negated by adding a not gate to the output signal of the NFA; however, because the not-gate is not part of the automaton formalism, further automaton operations such as concatenation would be undefined.

## 6.4 Debug Enhancements for Checkers

In their default form, assertion checkers provide feedback on violations through the assertion signal (result signal). In this section, different enhancements to checkers are explored, all of which share the goal of improving the debugging process. The debug enhancements introduced for assertion checkers are:

1. Reporting Signal Dependencies;
2. Monitoring Activity;
3. Signaling Assertion Completion;
4. Assertion and Cover Counters;
5. Hardware Assertion Threading.

The different techniques range from source-code comments (1) to actual modifications in the response of the checkers (3). Other enhancements constitute hardware additions and thus preserve the behavior of the checker outputs (2, 4 and 5). The overall goal is to increase the observability in the assertion monitoring in order to better assess the causes of a failure, or the causes of suspicious inactivity in a checker.

### 6.4.1 Reporting Signal Dependencies

When debugging failed assertions, it is useful to quickly determine which signals and parameters can influence the assertion output. In the generated checkers' HDL code, all of the signal and parameter dependencies are listed in comments before each assertion circuit. When an assertion fails, the signals that are referenced in an assertion can be automatically added to a wave window and/or extracted from an emulator, in order to provide the necessary visibility for debugging. Dependencies are particularly helpful when complex assertions fail, especially when an assertion references other user-declared sequences and/or properties, as allowed in PSL [111]. In such cases, assertion signal dependencies help to narrow down the causes of an error. Signal dependencies can also help to determine which signals must be stimulated in order to properly exercise an assertion that is found to be trivially true.

### 6.4.2 Monitoring Activity

The Booleans and sequences appearing in assertions can be monitored for activity to provide added feedback on the matching process. Monitoring the activity of a sequence is a quick way of knowing whether the input stimulus is actually exercising a portion of an assertion. The monitoring is performed on the states of the automata that represent the sequences and Booleans. Activity is defined as a disjunction of all states in an automaton. Anytime a state is active, the automaton is considered to be active. A sequence's automaton can exhibit internal activity when undergoing a matching, even if its output does not trigger. Conversely, if a sequence output triggers, the automaton representing it is guaranteed to show internal activity.

Using the appropriate compilation option, the checker generator declares activity signals for each sequence sub-circuit. The only states that are excluded from consideration for activity monitors are: initial states that have a `true` self-loop, and the final state when a sequence is the antecedent of the |=> operator. The reason for these exceptions is that any initial state with a `true` self-loop does not represent meaningful activity. Furthermore, when a sequence appears as the antecedent of a non-overlapped suffix implication, it is rewritten to an overlapped implication by concatenating an extra `{true}` sequence element to the end of the sequence, as shown in Section 5.4.2. This added sequence element creates a single final state in the antecedent, which also

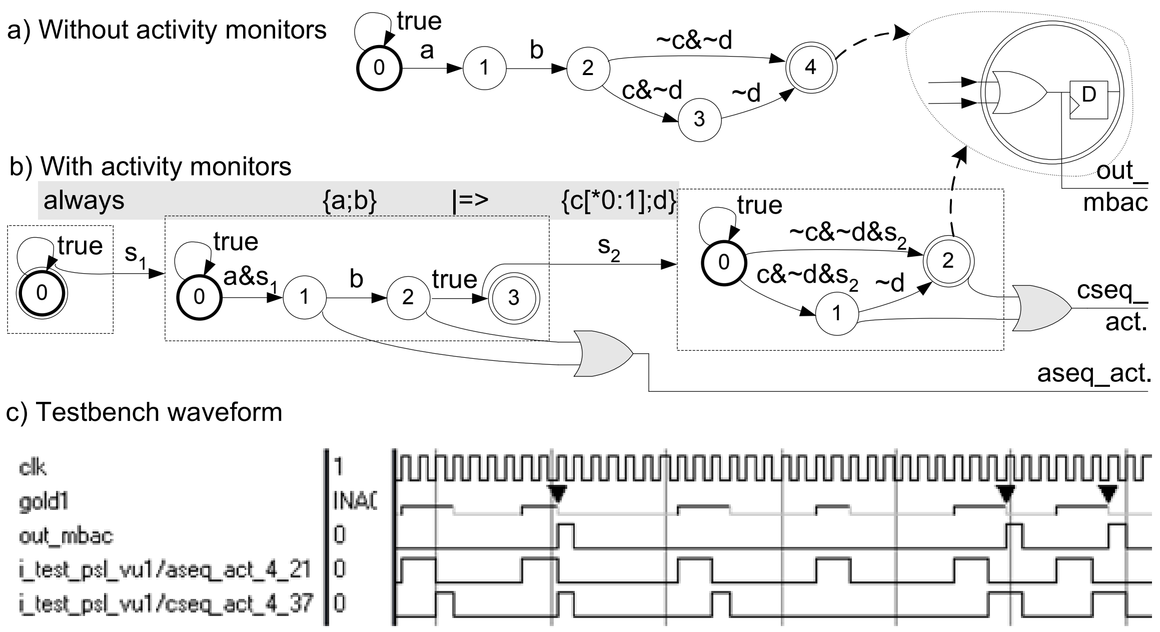


**Figure 6.3:** Activity signals for assertion: assert always ({$a$;$b$} |=> {$c$[*0:1];$d$});. The label *aseq* corresponds to the antecedent sequence, and *cseq* to the consequent sequence.

does not represent meaningful activity.

Under normal conditions, each assertion is represented by a single automaton before its transformation to RTL. To implement activity monitors, it is necessary to isolate the automaton of a sequence so that it is not merged with the remainder of the assertion's automaton during minimization. The automata that are isolated correspond to the sub-expressions that are being monitored, which in turn correspond to top-level sequences or Boolean expressions appearing in properties.

**Example 6.2:** The following assertion is used to illustrate activity monitors.

assert always ({$a$;$b$} |=> {$c$[*0:1];$d$});

Figure 6.3 a) shows how the example assertion normally appears as a single automaton when activity monitors are not desired. In Figure 6.3 b) activity monitors are produced, whereby the antecedent and consequent sequences must be kept isolated. The shaded OR-gates implement the disjunction of the state signals used to form the activity signals. The two types of exceptions that were stated previously apply in this example, namely that state 3 is not used in the antecedent's activity and state 0 is not used in the consequent's activity. It should also be stated that contrary to other signals that connect to the output of a state, the *out_ mbac* signal connects to the input of the flip-flop contained in the state, as was also shown in the example in Figure 4.9.

An example of activity traces is visible in Figure 6.3 c) for the example assertion. The activity signals for both sequences are visible, along with the assertion signal (*out_mbac*), and the assertion as interpreted by the ModelSim tool (*gold1*). As can be observed, the union of both activity signals coincides with ModelSim's activity indication. Since the checker's assertion signal is registered by default (not shown), it is asserted on the clock cycle following ModelSim's failure marker.

Monitoring the activity of a Boolean does not revert to monitoring the Boolean itself. In the following property

$$\textsf{always}\ a\ \textsf{->}\ \textsf{next}\ b$$

activity on $b$ is conditional to an activation from the antecedent $a$ occurring in the previous clock cycle.

Monitoring activity signals eases debugging by improving visibility in assertion-circuit processing. An implication in which the antecedent is never matched is said to pass vacuously (Definition 3.1). When monitoring antecedent activity, a permanently inactive antecedent does indeed indicate vacuity; however, this seldom occurs given that a single Boolean condition is enough to activate a state within the antecedent's automaton. An example to illustrate this is shown in Figure 6.3 b), where state 1 in the antecedent automaton is active every time $a$ is true.

In order for activity monitors to be the most effective for vacuity detection, the consequent needs to instead be monitored because an inactive *consequent* means that the antecedent was never fully detected, and thus never triggered the consequent. If no activity was ever detected in the consequent of a temporal implication, this indicates that the implication is vacuously true. The fact that the antecedent never fully occurred does not mean that there was no activity within it; conversely, activity in the antecedent does not mean that it fully occurred.

Activity monitors does not apply to sequences resulting from the application of rewrite rules. The rewrite rules are not implemented in true pre-processor fashion using syntactic replacement, and are implemented in the checker generator's internal functions. The checker generator actually creates new sub-trees in the syntax tree, corresponding to the rewritten expressions. The tool is therefore able to distinguish user-defined sequences from those resulting from rewrite rules.

```
1: FUNCTION: FirstMatch($\mathcal{A}$)
2: $\mathcal{A} \leftarrow$ StrongDeterminize($\mathcal{A}$)
3: remove all edges $(s_j, \sigma_j, d_j) \in \delta$ for which $s_j \in F$
4: return $\mathcal{A}$
```

**Algorithm 6.1:** First-match transformation algorithm.

### 6.4.3 Signaling Assertion Completion

For a verification scenario to be meaningful, the assertions must be exercised reasonably often. Assertions that do not trigger because the test vectors did not fully exercise them are not very useful for verification or debug. In cases where the assertions are trivially true, the true cause of a non-failure could be overlooked. On the contrary, assertions that are extensively exercised but never trigger offer more assurance that the design is operating as specified.

In the checker generator, assertions can be alternatively generated in *completion mode* to indicate when assertions complete successfully. This alternate assertion behavior can be very useful when assertion coverage is to be monitored, and to determine when assertions are not trivially true; these two factors that are important for creating effective test scenarios and testbenches. In general, by signaling successful witnesses, completion mode assertions provides an indication that if an assertion never failed, it was not because of a lack of proper stimulus.

Borrowing existing terminology [10], the completion mode identifies *interesting witnesses* to the success of a property. Vacuity, defined as antecedent non-occurrence in Definition 3.1, is only one possible cause for trivial validity. Completion mode can also be referred to as *pass checking*, *success checking* and *property coverage*.

Implementing completion mode requires the development of a first-match transformation algorithm. Creating an automaton that reports the first match, and only the first match of a sequence or a Boolean, for a given activation, requires strong determinization. Algorithm 6.1 shows the transformation algorithm used to produce a first-match automaton from a normal matching automaton. The completion mode transformation algorithm first determinizes the automaton such that each activation is represented by only one active state (line 2). From any given state, a deterministic automaton transitions into at most one successor state. The strong determinization algorithm from Algorithm 4.2 must be used, as opposed to the weak determinization algorithm. The determinization step is required so that when the first completion is identified, no other subsequent completions will be reported for the same activation. The second step in the algorithm removes all outgoing edges of the final states, when applicable (line 3). Any unconnected states resulting from this step are removed during minimization.

The completion mode affects assertions that place obligations on certain subexpressions, such as in the consequent of temporal implications for example. In temporal implications, for each observed antecedent, the consequent must occur or else the assertion fails. As opposed to indicating the first failure in each consequent, as is usually done, the completion mode assertion indicates the first success in the consequent. The completion mode has no effect on assertions such as

$$\textsf{assert never } s;$$

given that no obligations are placed on any Boolean expressions. This assertion states that the sequence argument $s$ should not be matched in any cycle. Every time the sequence is matched, a violation occurs and the assertion output triggers.

The actual elements of PSL that are affected by the completion mode are sequences and Boolean expressions when used directly as properties. More specifically, the completion mode is implemented by using the FirstMatch() algorithm in place of the FirstFail() and FirstFailStrong() algorithms appearing throughout Proposition 5.4 (properties). Strong sequences are treated as weak sequences in completion mode because the assertion signal, which now indicates successful completions, should not simultaneously be used to also indicate failures to comply with a strong operator. Implementing a completion mode with strong operators could be done using two separate output signals for assertion checkers, one indicating successful completions, and the other indicating failures of strong obligations at the end of execution.

**Example 6.3:** The following assertion is used to illustrate failure and completion mode automata:

$$\textsf{assert always } (\{a\} \mid\text{=>} \{\{c[\text{*}0{:}1];d\}|\{e\}\});$$

The assertion is normally compiled as the automaton shown in Figure 6.4 a), whereby the output signal (or final state) triggers each time the assertion fails. The completion mode automaton for this example is shown in Figure 6.4 b). When creating a completion mode checker, the automaton for the consequent of the implication is modified to detect the completion of the sequence, according to Algorithm 6.1. For a given activation (in this case $a$), only the first completion is identified by the automaton. The sequence of events $a; c; d$, makes the completion automaton trigger; however, the failure automaton does not reach a final state given that the sequence conforms to

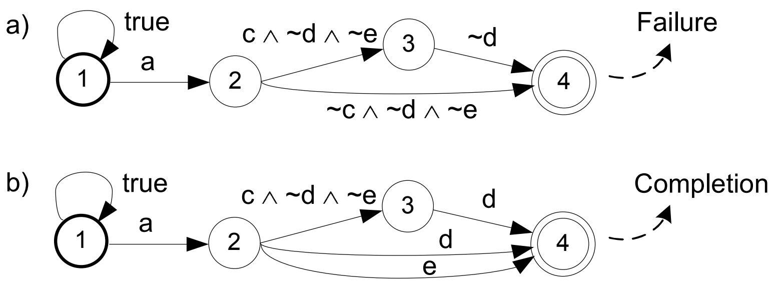


**Figure 6.4:** a) Failure and b) completion mode automata for the assertion in Example 6.3.

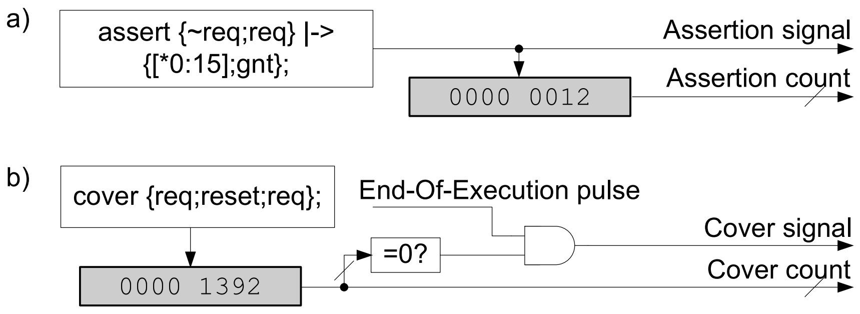


**Figure 6.5:** Counter circuits for: a) assertions and b) covers.

the specification indicated by the assertion.

The completion mode can be used with assertion counters to provide more detailed metrics in a test scenario.

## 6.4.4 Assertion and Cover Counters

The checker generator also includes options to automatically create counters on assert and cover statements for counting activity. Counting assertion failures is straightforward, as observed in the top half of Figure 6.5; however, counting the cover directive requires some modifications. In dynamic verification, cover is a strong property that triggers only at the end of execution. In order to count occurrences for coverage metrics, a plain matching (detection) automaton is built for the sequence argument, and a counter is used to count the number of times the sequence is matched. The cover's result signal only triggers at the end of execution if the counter is at zero, as shown in the lower half of Figure 6.5. If no counters are desired, a one-bit counter is implicitly used. The counters are width parameterized, and by threshold arithmetic do not roll-over when the maximal count is reached. The counters are also initialized by a reset of the assertion checker circuit.

Counters can be used with the completion mode from Section 6.4.3 to construct more detailed coverage metrics for a given test scenario. Knowing how many times an assertion completed successfully can be just as useful as knowing how many times an assertion failed. For example, if a predetermined number of a certain type of bus transaction is initiated, the related assertion should complete successfully the same number of times.

### 6.4.5 Hardware Assertion Threading

Assertion threading is a technique by which multiple copies of a sequence checking circuit are instantiated. Each copy (or thread) is alternately activated one after the other, as the sequence receives activations. This allows a violation's condition to be separated from the other concurrent activity in the assertion circuit, in order to help visualize the exact start condition that caused a failure. In general, using a single automaton mixes all the temporal checks in the automaton during processing. The advantage with this is that a single automaton can catch all failures; however the disadvantage is that it becomes more difficult to correlate a given failure with its input conditions. The assertion threading in effect separates the concurrent activity to help identify the cause of events leading to an assertion failure. Threading applies to PSL sequences, which are the typical means for specifying complex temporal chains of events.

By extension, threading applies to any PSL property in which one or more sequences appear. The threading of Booleans used as properties is not performed given the temporal simplicity of Booleans. The following property is used as a running example to illustrate the mechanisms behind assertion threading:

$$\textsf{always}\ a\ \texttt{->}\ \{b[*0{:}2];c\} \tag{6.4}$$

In this property the consequent sequence can last at most three clock cycles, and thus three threads will be used. For very long sequences, or even unbounded sequences that use the [*] operator (Kleene star), a larger but finite number of threads can be instantiated. When the number of threads is smaller than the maximum length of the sequence, it may not always be possible to completely separate the activity into different threads. If a thread is undertaking a matching and it receives another activation, identifying the precise cause of a failure becomes more difficult. When the resources allow it, increasing the number of hardware threads can help to properly isolate a sequence. In all cases, it is important to state that no matches can ever be

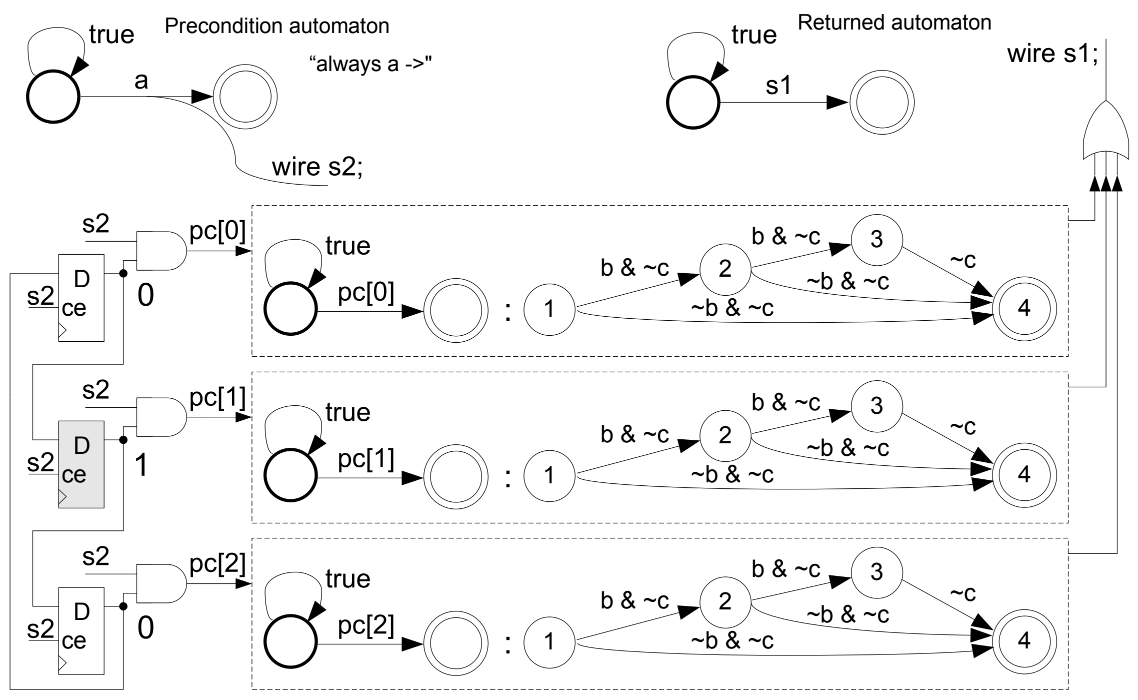


**Figure 6.6:** Assertion threading strategy. Ex: always $a$ −> $\{b[*0{:}2];c\}$.

missed, as a single automaton can concurrently handle all activations.

The assertion in (6.4) is threaded as shown in Figure 6.6. Although a property implication with a simple Boolean antecedent is used in this example, the sequence could also have been activated by more complex properties, such as a sequence antecedent when suffix implication is used, or complex Boolean events when the next_event_a property is used, for example. The first step in threading is to separate the sequence automaton from its precondition automaton. In this case the precondition automaton is a simple two state automaton with an edge for the Boolean $a$. Since $a$ is under the scope of the always operator, it is continually triggered, hence the self-loop with true in the initial state.

The second step in threading is to build the dispatcher, shown in the left side of the figure. The hardware dispatcher redirects the activations coming from the precondition to the multiple sequence-checking units in a round-robin manner. The dispatcher flip-flops form a one-hot encoded register such that each activation is sent to only one of the hardware threads. Each signal from the dispatcher ($pc[0]$ to $pc[2]$) activates its own copy of the sequence by using automata fusion, denoted ":" in Figure 6.6. The fusion is the same algorithm that was devised for SEREs in the previous chapter. The effect of the fusion with the small $pc[x]$ automaton is that each time a precondition signal is true, it will trigger the matching in the sequence automaton for the corresponding thread. The sequence automaton on the right side of fusion corresponds to the sequence that is to be threaded, which in this case is used as a property. The automaton for this sequence corresponds to FirstFail($\mathcal{A}$({$b$[*0:2];$c$})).

An alternate design choice for the dispatcher is also possible, by using a counter and a decoder. The counter requires $\lceil \log_2(n) \rceil$ flip-flops, compared to $n$ flip-flops in the first approach, where $n$ is the number of threads. Although the number of flip-flops is reduced in the counter approach, a decoder circuit is required, and for many threads a non-trivial amount of logic would be required. The first approach strikes a good balance between combinational logic and flip-flops, as is visible in the left side of Figure 6.6, and is well suited for FPGA implementations; the absence of decoding logic also improves its maximum operating frequency.

The result signals of the threads are combined in an OR-gate, such that if any thread fails, the property fails as a whole. The automaton that is returned is a small two-state automaton as shown at the top right in the figure. The symbol on the edge between the initial and final states is precisely the result signal from the OR-gate. The result signal of a thread's automaton is formed as shown in the top right portion of Figure 6.3 (as a disjunction of incoming edges, before the FF in the state). Threading also applies to the plain matching sequence automata, as opposed to the failure matching automaton discussed above.

Seen from the sub-circuit boundary, a multi-threaded sub-circuit's behavior is identical to that of a non-threaded sub-circuit. To implement threading, it is necessary to isolate a sub-automaton so that it is not merged with the remainder of the assertion's automaton during automata optimizations. Threading does not apply to sequences resulting from the application of rewrite rules. As mentioned, rewrite rules are not implemented using syntactic replacement in a pre-processor step and are instead implemented in the checker generator's internal functions. This way the tool is able to distinguish user-defined sequences from those resulting from rewrite rules.

An example scenario where assertion threading is useful is in the verification of highly pipelined circuits such as a CPU (Central Processing Unit) pipeline or a network processor, where temporally complex sequences are modeled by assertions. In such cases, it is desirable to partition sequences into different threads in order to separate a failure sequence from other sequences. Once temporally isolated, the exact cause of a failure can be more easily identified. The following example shows how assertion threading can be used to quickly identify the cause of an execution error in a CPU pipeline.

**Example 6.4:** In this example the design under verification is a simplified CPU execution pipeline, inspired by the DLX RISC CPU [97]. The execution unit has a five-level pipeline and executes instructions that perform memory and register manipulations. In the example, only memory writes and register writes are considered. An error injection mechanism is also incorporated in the instruction decoder, such that errors can be inserted for testing purposes. For a given write instruction only two possible destinations are allowed by the architecture, either the memory or the register file, but not both. The following block of PSL statements form an assertion that can be used to detect a faulty write operation.

**CPU_ASR**:

```
default clock = ( posedge Clk );
sequence Swr_instr = {InstrValid && (Instr[31:29]==3'b110 ||
                      Instr[31:29]==3'b101)};    //write instruction
sequence Smemwr = {[*2] ;  MemWr ; ~RegWr};      //memory write only
sequence Sregwr = {[*2] ; ~MemWr ; RegWr};       //register write only
assert always { Swr_instr } |=> { {Smemwr} | {Sregwr} }; //write works
```

Three sequences are declared, along with the default clock. The first sequence spans one clock cycle and specifies the Boolean conditions for the issuing of a write instruction. The next two sequences specify the proper behavior of memory and register writes. The assertion states that whenever a write instruction is issued, either a memory-only or register-only write should ensue. When a write instruction is issued, either of these two sequences should hold, hence the use of SERE disjunction in the consequent of the suffix implication.

In the memory and register write sequences, it can be observed that memory writes are committed in the fourth stage, and register writes in the fifth stage. The first stage is the issued instruction. The non-overlapping suffix implication and the [*2] prefix in the sequences cause the $MemWr$ signal to be tested in the fourth stage and the $RegWr$ to be tested in the fifth stage, as required.

The PSL statements are given to the checker generator to produce an assertion circuit, which is then instantiated in the CPU's RTL code. The checker runs in parallel with the CPU and monitors its signals for faulty executions. Figure 6.7 shows the resulting simulation trace, as exercised in a testbench. The "*Faulty instruction*" signal is used to create an error to illustrate how assertion threading can be used in the debugging process.

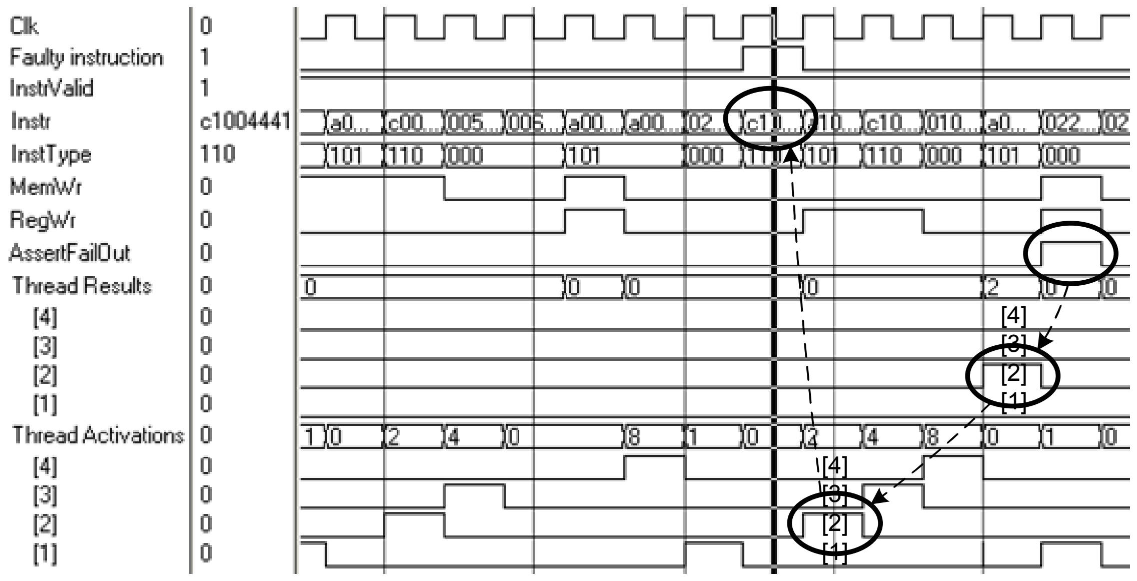


**Figure 6.7:** Use of assertion threading in the CPU example.

The *AssertFailOut* signal triggers at a given time point (at the right in the figure), thereby indicating that a violation occurred in the behavior of the write instruction. Since the assertion signal is registered, the thread that caused the error can be identified in the preceding cycle; in this case the error came from thread number two. Tracing back to the clock cycle where this thread was activated, and knowing that the instruction was issued in the previous cycle, it can be deduced that the faulty instruction occurred in the clock cycle where the cursor is located in the figure (which happens to correspond to the cycle in which the error was injected).

Assertion threading can be especially beneficial in more complex pipelines such as in superscalar processors, and even in non-processor designs where a large amount of concurrent activity is taking place.

## 6.5 Checkers in Silicon Debug and On-Line Monitoring

Circuit-level assertion checkers can be used not only for pre-fabrication functional verification, but also for post-fabrication silicon debugging, as illustrated in Figure 6.8 b). Assertion checkers can be purposely left in the fabricated circuit to increase debug visibility during initial testing of the device. Assertions compiled with a checker generator can also be used as on-line circuits for various in-field status monitoring tasks,

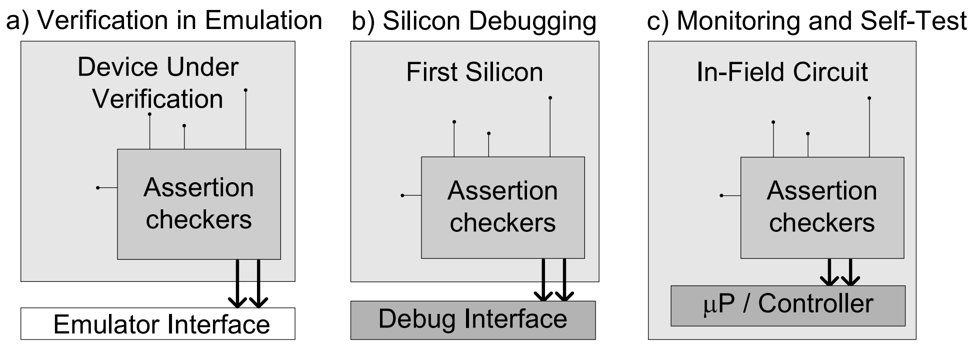


**Figure 6.8:** Usage scenarios for hardware assertion checkers.

as shown in Figure 6.8 c).

Section 6.5.1 expands on how assertion checkers can be used beyond the verification stages, and into full silicon debugging. Section 6.5.2 shows how assertions, combined with a checker generator, can be used to automatically design certain types of circuits. Example scenarios are shown in self-test and in-field monitoring applications. Section 6.5.3 introduces an algorithm for assertion grouping and shows how checkers can be managed in a dedicated reprogrammable logic core.

### 6.5.1 Checkers in Silicon Debugging

Assertion checkers produced by the checker generator can not only be used for emulation and simulation verification before fabrication, but can also be used post-fabrication, when a set of assertion checkers is purposely left in the design. The goal of the silicon debugging process is to find and possibly correct design errors in a post-fabricated Integrated Circuit (IC), usually referred to as *first silicon*. The checkers can test for functional faults and timing issues which can not be fully tested pre-fabrication. By connecting the checker outputs to the proper external test equipment or on-board read-back circuits, the user can get immediate feedback on assertion failures in order to undertake the debugging process. A checker generator capable of producing resource-efficient checkers is clearly an advantage when checkers take up valuable silicon area in the device.

Assertion-based silicon debug differentiates itself from emulation based verification because in silicon debug, the design is implemented in its intended technology, as opposed to being implemented in reprogrammable logic during hardware emulation. This allows at-speed debugging under expected operating conditions, and assertion checkers play an important role here as well. Figure 6.9 a) shows how assertion checkers in silicon are used to monitor the state of the device under test during the

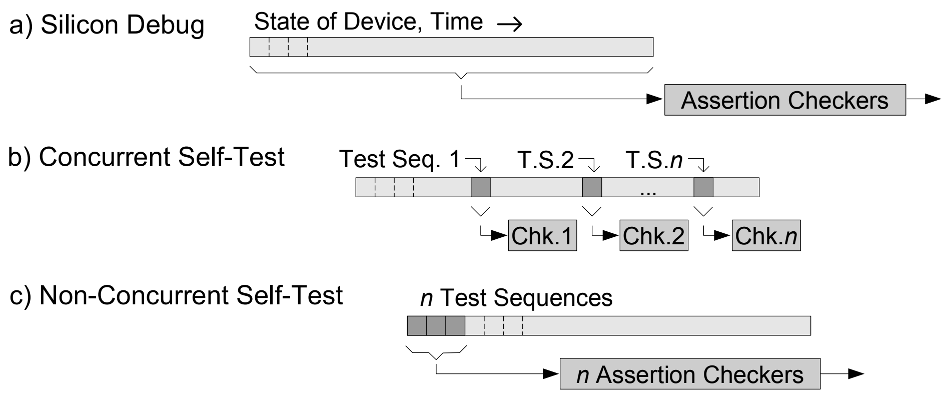


**Figure 6.9:** Debugging and self-test using checkers.

entire execution. This monitoring mode is identical to that which is used verification, with the nuance that the checkers exist in permanent silicon and can be used during the lifetime of the device, as opposed to temporary verification checkers that are removed before fabrication.

### 6.5.2 In-Circuit On-Line Monitoring

The checkers for silicon debugging mentioned above serve their purpose, but can ultimately be removed for mass production of the IC. In a more general usage scenario, the expressive power of assertions, combined with a checker generator can be used to actually perform explicit circuit design, going beyond the bounds of verification and debugging. In this view, any form of monitoring circuit that can be expressed by an assertion, once fed into the checker generator, can produce a complex error-free circuit instantly. These circuit-level checkers are in fact more akin to actual design modules rather than verification modules.

A checker generator allows the flexibility of automatically generating custom monitor circuits from any assertion statement. Coding checkers by hand can be a tedious and error-prone task. In certain cases, a single PSL statement can imply tens or even hundreds of lines of RTL code in the corresponding checker. Using assertions and a checker generator can be a very efficient way of automating the design of certain types of circuits. An example where this technique can be utilized is in designing certain portions of self-test circuits. Off-line Built-In Self Test (BIST) techniques are well established [3], and are based on the traditional flow:

$$\text{TPG} \rightarrow \text{CUT} \rightarrow \text{ORA}$$

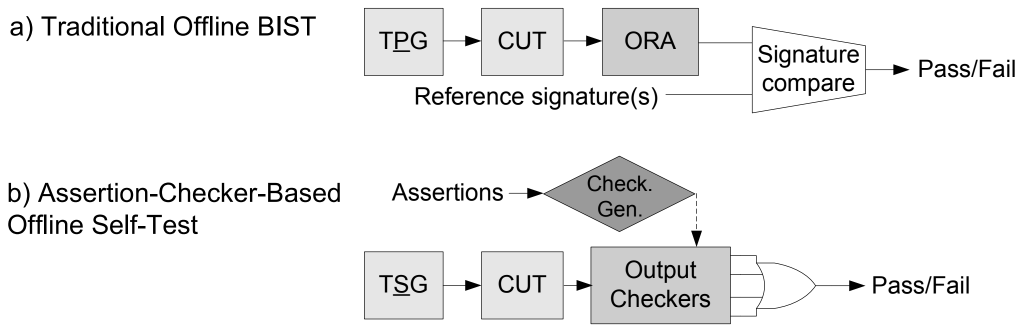


**Figure 6.10:** Traditional BIST vs. self-test using checkers. TPG = Test Pattern Generation, TSG = Test Sequence Generation, ORA = Output Response Analyzer.

where TPG symbolizes Test Pattern Generation and ORA is the Output Response Analyzer. This architecture is shown in Figure 6.10 a). Off-line BIST techniques typically employ a mixture of pseudo-random and deterministic TPG.

Assertion checkers can also benefit the implementation of self test circuitry, albeit at a higher level. Test pattern generation is instead referred-to as test sequence generation (TSG). Figure 6.10 b) shows an assertion-based off-line self-test architecture, whereby test sequences are applied to the input of the Circuit Under Test (CUT), and assertion checkers are used as the response analysis circuit. In this approach the signature can be encoded as one bit, representing success or failure. The offline self-test, when executed prior to device startup is considered non-concurrent, and is illustrated in Figure 6.9 c). The use of assertions and a checker generator allows the response analysis circuitry to be designed with greater ease.

Checker-based techniques also apply to the design of on-line self-test circuits [4], as shown in Figure 6.9 b). In this scenario, the checker generator is used to design the analysis circuits that correspond to the given test sequences. Contrary to silicon debug and the other self-test techniques mentioned previously, a checker for a given test sequence is only used as a response analyzer when the test sequence is being exercised. In the concurrent self-test model, the device is momentarily interrupted for testing, or alternately, unused resources are concurrently tested during runtime.

Using assertions and a checker generator as a means of circuit design poses difficulties when it comes to generation signals; however, the design of many types of monitoring and analysis circuitry can benefit directly from this technique. The high-level expressiveness of an assertion language combined with a checker generator can be used as a quick method to automatically design circuits.

If checkers are incorporated in the final design, in-circuit diagnostic routines that

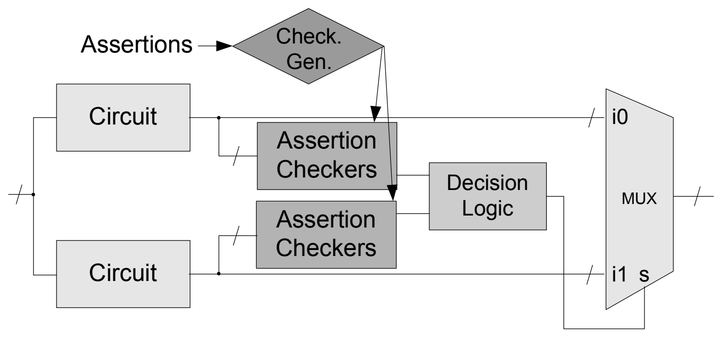


**Figure 6.11:** Run-time status monitoring using assertion checkers for redundancy control.

utilize the checker output can be implemented. Assertion checkers can be an integral part of any design that attempts to assess its operating conditions on-line in real time. Run-time assertion monitoring can be performed by the checkers, and the results can be analyzed by an on-board CPU that can then send regular status updates off-chip. Real-time monitoring based on in-circuit checkers can be especially important in mission critical environments. For example, if a multitude of assertion checkers running concurrently with the device were to detect an error, a secondary redundant system could be instantly activated. Figure 6.11 shows an example of how checker generator can be used to design the monitoring circuits for switching in redundant systems. Designing an array of safety-checking circuits can be more easily performed using assertions and a checker generator.

### 6.5.3 Assertion Grouping

With increasing device complexity and the advent of System-on-Chip (SoC) and Network-on-Chip (NoC) designs, small blocks of reprogrammable logic are commonly added to ICs. This allows a certain amount of flexibility for correcting post-fabrication defects, and also allows a certain amount of further design modifications to be performed. When a large amount of assertion checkers is to be instantiated in a design, space constraints can limit their applicability. If the device utilization is near maximum capacity, there may not always be room for all the checkers. Furthermore, if the device has the necessary free space for adding a small area of reprogrammable logic (if not already present), and if the logic can be connected to the main signals of interest throughout the device, then in such cases *assertion grouping* must be performed to manage a large set of checkers. In assertion grouping, the set of checkers is partitioned into groups whereby each group is guaranteed not to exceed the size of

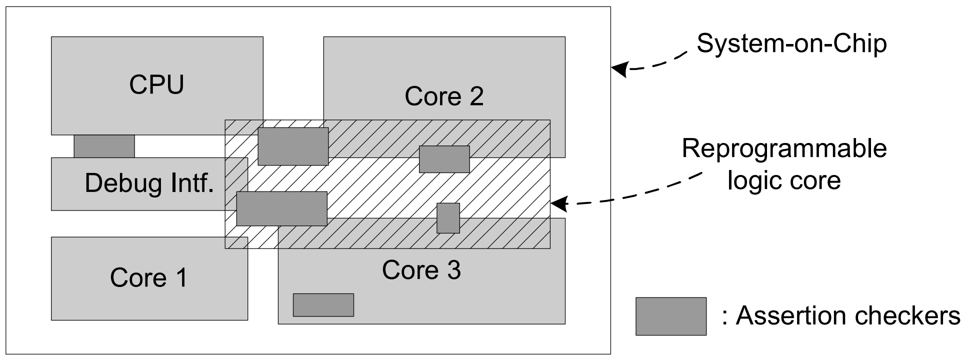


**Figure 6.12:** Fixed and reprogrammable assertion checkers in SoCs.

the reprogrammable logic area. Ideally, the number of groups should be minimized so that fewer reconfiguration steps are necessary.

In SoC designs, some modules are often carried over from a previous design and do not require a detailed verification. Newly designed modules will require greater interoperability with the reprogrammable fabric, in order to perform assertion-based debugging. Protocol monitors may also be required to debug complex interactions between cores. The reprogrammable logic can also be used for assertion checking in offline unit testing before the device is fully operational.

Figure 6.12 shows an example scenario where a reprogrammable logic core was added to a SoC. In this example, the CPU and Core 1 were part of a previous design and were known to be bug-free. Consequently, their interconnection with the reprogrammable logic is rather limited. Cores 2 and 3 are newer modules and have more connections to the logic, in the event that circuit corrections need to be performed in the fabricated IC. In the example figure, two assertion checkers are placed in permanent silicon, and four are temporarily programmed in the reprogrammable fabric.

The reprogrammable logic core can even be used to perform the assertion-based concurrent BIST described in Figure 6.9 c). In this scenario, the CPU can coordinate the instantiation of the proper checkers for each test set in the reprogrammable fabric. Checker groups (also called partitions, or subsets) are instantiated one after the other in the reconfigurable area, corresponding to the set of test sequences being executed. Reprogramming reconfigurable logic on-the-fly for different tasks is known as run-time reconfiguration [150].

The checker generator typically processes a set of PSL statements and transforms them into an RTL module of synthesizable code. In the generated checker, each assertion circuit's RTL code is clearly marked. A set of Python scripts was developed

```
1: FUNCTION: SUBSET-CIRCUIT(set $C$ of circuit metrics (FF, LUT), $area_{FF}$,
   $area_{LUT}$)
2: $D \leftarrow C$
3: while there are circuits left in $C$ do    //phase 1 (dominant metric is #FFs)
4:    sort circuits $C$ according to $\#FF$s
5:    build dynamic programming table $T$ for subset-sum on $\#FF$s
6:    search $T$ for best subset $S$ such that $\sum_{s_i \in S} \#LUTs(s_i) < area_{LUT}$
7:    log subset circuits in $S$ as a group in phase 1 results
8:    remove circuits $S$ from $C$
9: while there are circuits left in $D$ do    //phase 2 (dominant metric is #LUTs)
10:   sort circuits $D$ according to $\#LUT$s
11:   build dynamic programming table $T$ for subset-sum on $\#LUT$s
12:   search $T$ for best subset $S$ such that $\sum_{s_i \in S} \#FFs(s_i) < area_{FF}$
13:   log subset circuits in $S$ as a group in phase 2 results
14:   remove circuits $S$ from $D$
15: if number of subsets in both phases differs then    //analysis
16:   return results of phase which has the fewest subsets (groups)
17: else
18:   return results of phase for which the subset-sum was performed on metric with
      smallest freedom
```

**Algorithm 6.2:** Assertion circuit partitioning algorithm.

to extract each assertion circuit from the checker's RTL module, and to automate the individual synthesis of these checkers. This is performed so that circuit-size metrics can be obtained for each checker. Another script is responsible for logging these metrics in a file, which is then used as an input to the partitioning algorithm, described next. FPGA synthesis tools are used in this example scenario.

Once the checkers have been individually synthesized and their sizing metrics are obtained, the partitioning algorithm shown in Algorithm 6.2 is used to create subsets of checkers suitable for multiple reconfigurations in the reprogrammable logic area. This algorithm is based on solving the subset-sum problem by dynamic programming [56]. However, because the circuit metrics comprise two variables, namely # of flip-flops (FF) and # of lookup tables (LUT), the typical subset-sum procedure can not be employed directly on its own. A two-phase algorithm is developed, which returns a near-optimal partition. The inputs of the algorithm are the circuit metrics and the size of the reprogrammable area (also specified as # of flip-flops and # of lookup tables).

Phase 1 in the algorithm (lines 3 to 8) uses flip-flops as the dominant metric and performs a subset-sum computation on this metric (line 5). The subset-sum algorithm requires that the circuits be sorted in increasing order according to the dominant metric (line 4). A search is then performed for the best subset according to the size limit of this dominant metric, while also respecting the maximum size for the secondary metric (line 6). Once the best subset has been determined, it is logged and removed from the set (lines 7 and 8). This procedure continues until the set of checkers is empty (line 3).

The dominant and secondary metrics are interchanged and the same procedure is repeated (lines 9 to 14). A comparison is then made between both phases (lines 15 to 18), and the solution with the fewest subsets is logged. When both phases have the same number of subsets, it was empirically observed that the more balanced partition is the one for which the dominant metric is the most constrained by the area limits (smallest freedom).

It can be shown by counterexample that the algorithm is not guaranteed to create an optimal partition; however, experiments show that it drastically outperforms the brute force approach in computation time. Furthermore, when one of the metrics has a large amount of freedom with respect to its area constraint, the problem tends toward a single variable subset sum for which the algorithm is optimal.

# Chapter 7

# Evaluating and Verifying Assertion Checkers

## 7.1 Introduction and Overview

In this chapter the assertion checkers produced by the MBAC checker generator are empirically evaluated. The MBAC tool incorporates the techniques introduced in the three previous chapters, and is coded in C++. MBAC is a standalone executable invoked at the command line. The experiments performed herein have the goal of evaluating three principal factors associated with assertion checkers:

1. Hardware resource usage and operating frequency of checkers;
2. Functional correctness of checkers;
3. Support of PSL operators in the checker generator.

The checkers should utilize the fewest circuit primitives as possible and should run at the highest possible clock speed. Resource efficient checkers are important so that the in-circuit debugging capabilities resulting from the ABV methodology can be performed less obtrusively when checkers are added to a circuit under test. The small circuit size of checkers is also beneficial when the checkers are used as permanent status monitors, and more generally when checkers are used as actual design modules. The checkers should also correctly implement the behavior specified by their respective assertions, and the checker generator should support all the PSL operators in the simple subset.

In the majority of the experimental results, the checkers produced by MBAC are compared to the FoCs checker generator from IBM, which is the only available stand-alone checker generator. The versions of the tools that are the most recent at the time of writing are MBAC version 1.75 and FoCs version 2.04 [108]. When evaluated in hardware, the checkers are synthesized with Xilinx XST 8.1.03i for an XC2V1500-6 FPGA, and the synthesis is optimized for speed (as opposed to area). The number of FFs and lookup tables (LUTs) required by a circuit is of primary interest when assertion circuits are to be used in hardware emulation and silicon debugging. Because speed may also be an issue, the maximum operating frequency after synthesis for the worst clk-to-clk path is reported. Although in this technology the fundamental metrics are the number of Flip-Flops (FF) and four-input lookup tables (used for implementing combinational logic), the checkers could also be synthesized in other technologies such as standard cell ASICs (Application Specific Integrated Circuits).

When a vunit contains multiple assertions, an HDL module containing the checkers is created. A vector of output signals is declared, where each bit in the output vector corresponds to an assertion signal. For synthesis metrics to be meaningful, each assertion checker must be synthesized individually to avoid any resource sharing at the hardware level. This is accomplished by redefining the output vector as a one-bit signal, and synthesizing the module multiple times, whereby the desired assertion signal is routed to the output. In this way, any unused logic is trimmed. The only exception to this is when groups of assertion checkers are synthesized, as in Section 7.3 for example, where assertion grouping is evaluated.

When evaluated in software, the RTL checkers are simulated in the ModelSim simulator from Mentor Graphics (version 6.1f SE) [129], where the behavior of the checkers can be compared to the assertions as interpreted by ModelSim.

Functional equivalence of checkers can also be verified formally using model checking. This procedure can be used to assess whether two checkers for the same assertion are actually behaving the same way, and will be used in this chapter both to compare MBAC checkers with FoCs, and to compare the two different implementations of checkers for the same assertion in the MBAC tool.

The model checking technique for performing sequential equivalence checking is illustrated in Figure 7.1. The model checker used in the experiments is Cadence SMV [43]. The central element is the exclusive-or gate that evaluates to logic-1 when both inputs are not equal. The additional inverter is used because the assertion polarity of FoCs' circuits is different than with MBAC's circuits. The outputs of MBAC's checkers are normally at logic-0 and momentarily rise to logic-1 to signal assertion errors; the opposite polarity is used in FoCs. A testbench instantiates both checkers and postulates that the output of the xor gate is always at logic-0 (i.e. the circuits are equivalent). The assertion statement used to specify the equivalence has

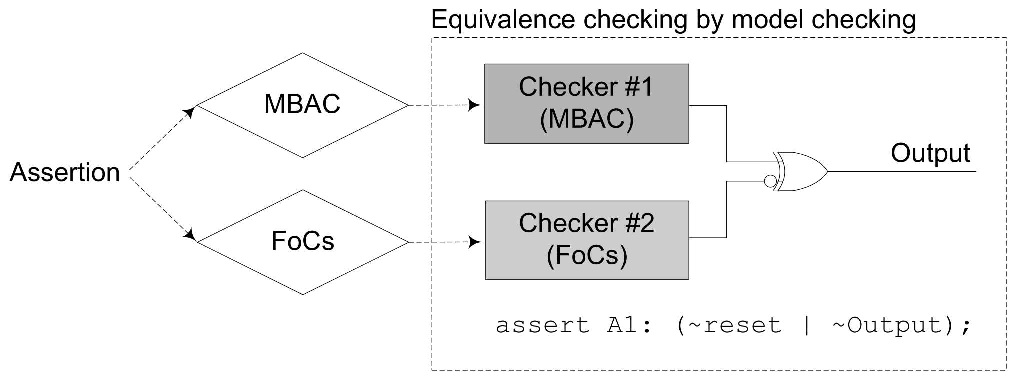

Equivalence checking by model checking
Assertion
MBAC
FoCs
Checker #1 (MBAC)
Checker #2 (FoCs)
Output
assert A1: (~reset | ~Output);

**Figure 7.1:** Equivalence checking by model checking.

the form:

$$\textsf{assert}\ reset \text{ -> } {\sim}XOR\_Output;$$

where the reset signal is active low. The assertion above states that when the reset is not active (i.e. logic-1), then the output of the xor gate should always be at logic-0. In the Cadence SMV model checker, the assert directive, when placed in an always procedural block, specifies a design invariant and is checked in all possible executions. This means that no temporal operators need to be used, such as G in LTL or AG in CTL, to ensure that the property holds globally (always) and for all possible executions.

In Figure 7.1, the implication is rewritten using disjunction as shown in (5.1), such that it can be understood by the model checker. When the circuits are functionally equivalent, the model checker returns *true*, otherwise it returns *false* and produces a counterexample. It should be emphasized that this is a static verification with implicit complete coverage, and no test vectors need to be supplied.

In the following two sections, non-synthetic assertions are used to evaluate the checker generator. The term *non-synthetic* is used to refer to real assertions appearing throughout the literature and related research. This is in contrast to synthetic assertions appearing in the remainder of the chapter. As will be explained further, common real-world assertions are not temporally complex enough to truly benchmark the effectiveness of a checker generator, and a set of hand-designed assertions over plain Booleans will also be devised.

In the experiments, the actual source design is not implemented and the emphasis is placed on the actual checkers that are generated from the assertions. For example, in the PCI-read assertion from the book “Assertion-Based Design” [74], the PCI interface to which the assertion refers to is not given, and is not required. Incorporating the entire source design in the circuit metrics does not allow the size of the checkers to be discerned properly, and diverts attention from the size of the checkers themselves. Furthermore, the source design could be coded in a variety of ways, some more efficient than others, and would not represent a universal metric. In general, an upper bound on the hardware overhead associated with assertion checkers can be obtained by adding the size of the checkers to the size of the source design. When the checkers are synthesized with the source design however, resource sharing helps to reduce the total circuit size. This resource sharing can occur between the checkers and the source design, or between the checkers themselves.

When the source design is not available, all that must be done for the checker generator to function correctly is to define a *declaration module* containing only the declarations of the signals used in the assertion. This HDL module does not have any actual functionality, and is the minimal requirement for the MBAC and FoCs checker generators to operate properly. The `vunit` containing the assertions is then bound to the declaration module.

In all experiments, the outputs of the checkers are sampled by a flip-flop (FF) sensitive to the rising edge, such that equivalence checks and simulation results are glitch-free. In some cases in equivalence checking, the flip-flop is made to be a monostable flip-flop that perpetually remains in the triggered state once the assertion signal has triggered. This does not affect the validity of the equivalence check and is used when the run-time semantics of two checkers are different, while still both being correct, as discussed in Section 2.2 regarding (2.3). When the monostable flip-flops are used at the outputs of the checkers, a verified equivalence indicates that both circuits are identical when the pre-FF output does not trigger, and indicates that both find the first error at the same clock cycle, for all possible error traces. Once an assertion has failed, it has failed as a whole, and one could in fact stop reporting errors altogether, although this would not be ideal for debugging purposes.

The majority of the experimentations compare the checkers produced by MBAC to those produced by the FoCs tool after synthesis to FPGA technology. Exceptions to this are Sections 7.4 and 7.5. In Section 7.4, the checkers are compared by evaluating the RTL code before synthesis. This reveals insight into the size of the checkers that are generated in HDL, before any of the optimizations brought upon by the synthesis tool are applied. In Section 7.5, the debugging enhancements for hardware assertion checkers that were introduced in the previous chapter are novel and can not be compared to any other tool.

## 7.2 Non-Synthetic Assertions

In this section, non-synthetic, or real-world assertions, are used to evaluate the checker generator. The non-synthetic assertions are meant to verify particular aspects of real design modules, and their Boolean layer expressions do not typically consist purely of simple signals. In the majority of test cases, a declaration module is created to represent the CUT, where only the signal declarations are defined. The simplified Booleans have the advantage of more clearly revealing the temporal complexity of the checkers, since the implied hardware is not bloated with combinational logic for complex Boolean expressions.

The assertions used in the test cases are shown below, and the synthesis results and the comparisons to the FoCs checkers are presented at the end of the section. The default clock declarations are omitted, and all signals are sensitive to the rising edges of the clock. In all cases an active low reset was used. The vector widths of signals are specified, and all signals that are not mentioned are assumed to be single-bit entities.

The first assertion used to evaluate the checker generator is the PCI assertion adapted from page 102 of Foster's book [74]. This assertion models the correct behavior of a read transaction for the PCI bus, and is specified using multiple sequence declarations. Different phases of the transfer are modeled separately, such as turn-around and the address and data phases. The *cbe_n* signal is three bits wide.

**PCI_ASR**:

```
‘define IO_READ 4’b0010
‘define MEM_READ 4’b0110
‘define CONFIG_RD 4’b1010
‘define MEM_RD_MULTIP 4’b1100
‘define MEM_RD_LINE 4’b1110
---
‘define data_complete ((!trdy_n || !stop_n) && !irdy_n && !devsel_n)
‘define end_of_transaction (‘data_complete && frame_n)
‘define adr_turn_around (trdy_n & !irdy_n)
‘define data_transfer (!trdy_n && !irdy_n && !devsel_n && !frame_n)
‘define wait_state ((trdy_n || irdy_n) && !devsel_n)
‘define cbe_stable (cbe_n==prev(cbe_n))
‘define read_cmd ((cbe_n == ‘IO_READ) || (cbe_n == ‘MEM_READ) ||
  (cbe_n == ‘CONFIG_RD) || (cbe_n == ‘MEM_RD_MULTIP) ||
  (cbe_n == ‘MEM_RD_LINE))
```

```
---
sequence SERE_RD_ADDR_PHASE = {frame_n;!frame_n && ‘read_cmd};
sequence SERE_TURN_AROUND = {‘adr_turn_around};
sequence SERE_DATA_TRANSFER = {{‘wait_state[*];‘data_transfer}[+]};
sequence SERE_END_OF_TRANSFER = {‘data_complete && frame_n};
sequence SERE_DATA_PHASE = { {{SERE_DATA_TRANSFER} ;
                      {SERE_END_OF_TRANSFER}} && {‘cbe_stable[*]} };
---
property PCI_READ_TRANSACTION = always ({SERE_RD_ADDR_PHASE} |=>
                  {SERE_TURN_AROUND; SERE_DATA_PHASE}) abort !rst_n;
assert PCI_READ_TRANSACTION;
```

The only change done in the assertion above concerns the addition of the [*] repetition in the right side of the intersection (&&) in the SERE_DATA_PHASE sequence. When the assertion is used as presented in the Foster book [74] (i.e. without the [*]), the checker generator produces a suspiciously small checker where the majority of the Boolean layer signals are not used. This reveals a potential error in the assertion itself, and upon closer inspection, it can be seen that when the ‘*cbe_stable* sequence is not repeated, the length matching intersection in the SERE_DATA_PHASE can never produce a match. This is because the ‘*cbe_stable* is a sequence that spans one cycle, and the left side of the length-matching && spans at least two clock cycles. The corrected version of the assertion is used in the experiment, and the original one likely contained a typographical error.

The next two assertions are used to ensure correctness of the AMBA bus protocol, and are from Chapter 8 in “Using PSL/ Sugar for Formal and Dynamic Verification” [53]. Each assertion represents the most temporally complex assertion from the two main AMBA test suites, each comprising a set of 26 assertions. The first assertion, AHB_ASR, captures the requirement that a bus transaction must never stall for more than 16 cycles. The *hResp* signal is a two-bit vector indicating the outcome of the transaction (other outcomes are ERROR, RETRY and SPLIT).

**AHB_ASR**:

```
localparam OKAY = 2’b00;
---
property NeverMoreThan16WaitStates = always (
  {(hReady == 1);(hReady == 0)} |=>
```

```
    {{(hReady == 0)[*0:15]};{hReady == 1}}
    abort (hResp != OKAY));
assert NeverMoreThan16WaitStates;
```

The second AMBA bus protocol assertion verifies the correct operation of the state machine used to decode a read transaction, as seen from a bus slave interface. The *fsm_read* and *hburst* vectors are three bits wide, and *htrans* is two bits wide. The *localparam* keyword was changed to *parameter* in the test files used.

**MemSlave_ASR**:

```
localparam NONSEQ = 2'b10;
localparam SINGLE = 3'b000;
localparam FSMR_IDLE    = 3'b000;  // no ahb activity
localparam FSMR_READ_P1 = 3'b001;  // ahb in read mode, pipe 1
localparam FSMR_READ_P2 = 3'b010;  // ahb in read mode, pipe 2
localparam FSMR_READ_P3 = 3'b011;  // ahb in read mode, pipe 3
---
sequence qNonSeqReadTransferSingle = {hburst==SINGLE &&
  htrans==NONSEQ && hwrite==1'b0 && hsel_0n==1'b0 && done_xfrn==1'b0};
--
property ReadFsmPipelineSingle = always(
  {qNonSeqReadTransferSingle} |=>
         {fsm_read==FSMR_READ_P1 && hready==1'b0;
          fsm_read==FSMR_READ_P2 && hready==1'b0;
          fsm_read==FSMR_READ_P3 && hready==1'b1;
          (fsm_read==FSMR_IDLE)});
assert ReadFsmPipelineSingle;
```

The Open Verification Library (OVL) [75] consists of a set of parameterized HDL assertion checkers, for use in various scenarios from general (ex: always and never checkers) to particular (ex: parity and FIFO checkers). Version 2.0 beta of the OVL was used in the experiments. Here also, the most temporally complex assertions were selected, and in this case four assertions were retained for benchmarking. The *test_expr*2 signal is *width* bits wide (i.e. [*width*-1:0]).

**OVL_ASRa - OVL_ASRd**:

```
parameter num_cks = 2;
parameter max_cks = 2;
parameter width = 8;
---
property ASSERT_FRAME_RESET_ON_START_MAX_CHECK_P = always (
         ({(max_cks > 0) && reset_on_new_start &&
          rose(start_event) && !test_expr} |=>
         {!test_expr[*0:1];(test_expr || rose(start_event))})
         abort(!reset));
assert ASSERT_FRAME_RESET_ON_START_MAX_CHECK_P;
---
property ASSERT_HANDSHAKE_ACK_WITHOUT_REQ_SUBSEQUENT_REQ_P = always (
         ( {fell(ack)} |-> {{[*];rose(req);[*]} && {!ack[*];ack}} )
         abort(!reset));
assert ASSERT_HANDSHAKE_ACK_WITHOUT_REQ_SUBSEQUENT_REQ_P;
---
property ASSERT_UNCHANGE_RESET_ON_START_P = always (
         {(num_cks > 0) && reset_on_new_start && start_event} |=>
          {(test_expr2 == prev(test_expr2))[*]; !window}
          abort(!reset));
assert ASSERT_UNCHANGE_RESET_ON_START_P;
---
property ASSERT_WIN_CHANGE_P = always ( ({start_event && !window ;
         stable(test_expr2)[*1:inf]} |-> {!end_event})
         abort(!reset) );
assert ASSERT_WIN_CHANGE_P;
```

Another real-world assertion comes from the area of Network-on-Chip (NoC) applications, where an entire system and its network infrastructure are contained on the same integrated circuit. The test assertion used is from a Network-On-Chip application [48], and is used to ensure that a network message (flit) reaches its intended destination node. The assertion is parameterized through a set of register variables, such that it can be reprogrammed by the on-board CPU to track different flits. The network architecture is a Ring-of-Rings structure [35], comprised of one central ring and four satellite rings. The assertion ensures that a triple of flits is observed in the proper order in an interface. The *x_SrcGlobal* and *x_SrcLocal* signals are four-bit signals, and the $x\_Data$ signals are 32 bit vectors ($x\_DataValid$ signals are single-bit).

**NOC_ASR**:

```
sequence NR_P1 = {NR_DataIng_p_DataValid     == 1
  && NR_DataIng_p_SrcGlobalRing == Reg_SrcGlobal1
  && NR_DataIng_p_SrcLocalRing  == Reg_SrcLocal1
  && NR_DataIng_p_Data          == Reg_SrcData1};
sequence NR_P2 = {NR_DataIng_p_DataValid     == 1
  && NR_DataIng_p_SrcGlobalRing == Reg_SrcGlobal2
  && NR_DataIng_p_SrcLocalRing  == Reg_SrcLocal2
  && NR_DataIng_p_Data          == Reg_SrcData2};
sequence NR_P3 = {NR_DataIng_p_DataValid     == 1
  && NR_DataIng_p_SrcGlobalRing == Reg_SrcGlobal3
  && NR_DataIng_p_SrcLocalRing  == Reg_SrcLocal3
  && NR_DataIng_p_Data          == Reg_SrcData3};
---
assert always {NR_P1} |-> eventually! {NR_P2; [*]; NR_P3};
```

A large amount of assertions is used in the PROSYD project [107] to verify a data receiver block. The block is connected to a consumer and a producer, and various forms of assertions are explored, some which are equivalent to others. Five of the most temporally complex assertions are reported below and are used in the evaluation. The STATE vector is a two-bit enumerated type, RESERVED is an 8-bit vector, and DATA_IN and DATA_OUT are 64-bit vectors.

**DATARX_ASRa - DATARX_ASRe**:

```
assert never {END; {(!START)[*]; START; true} && {ERROR[=1]}};
---
assert { [*]; { rose(ENABLE) && STATE==active1; (rose(ENABLE) &&
        STATE==active1)[->]} &&  {START[=0]}} |=> {[*0:2] ; START};
---
assert always ((STATE==data && DATA_IN[63] &&
  (DATA_IN[7:0] == RESERVED[7:0]) && !STATUS_VALID) ->
    next_e[1:3](STATUS_VALID && (DATA_OUT[41] ||
      DATA_OUT[34] || DATA_OUT[37])));
```

```
---
assert {[*]; (REQ && STATUS_VALID)[=3]; STATUS_VALID[->]} |-> {ACK};
---
assert {[*]; {READ[=3]} && {WRITE[=2]}} |=>
       {(!READ && !WRITE)[*]; READY}! ;
```

Other non-synthetic assertions appearing throughout the previous chapters are also used to evaluate the checker generator. The up-down counter from Example 2.7 on page 45 contains four assertions respectively labeled UDCNT_ASRa to UDCNT_ASRd. The full HDL and PSL code of this example is given in Appendix A. The default width is assumed for the counter output and load buses (eight bits). The assertion contained in the CPU-write case study for assertion threading on page 147 (Example 6.4) is used, and is labeled CPU_ASR. The arbiter assertion in (2.1) on page 14 is also used, and is labeled ARB_ASR.

Table 7.1 shows the hardware synthesis results of the checkers for each assertion mentioned in this section, as produced by both the MBAC and FoCs checker generators. It should be noted that for assertions OVL_ASRd and UDCNT_ASRa, the stable() function is not supported by FoCs and was rewritten as shown in Proposition 5.1. Furthermore, the consequent of the suffix implication in UDCNT_ASRa and the argument of never in UDCNT_ASRd were braced {. . .}, as FoCs does not seem to support the non-braced versions. In all of these cases, FoCs did not support the original assertions and could have been marked as *not supported* in the table.

For the AHB_ASR assertion, the equivalence check failed and the counterexample below reveals a slight problem with the FoCs checker:

$\text{CE}_1$: { $\mathit{hready} \wedge (\mathit{hresp}\,!{=}\text{OKAY})$ ; [*17] }

In the counterexample above, signals not listed are assumed to be at logic-0, and in the 17 cycles symbolized by [*17], all signals are at logic-0, including *hresp*. In the assertion, by operator precedence the abort applies only to the consequent sequence (as opposed to the entire suffix implication). It is likely that in FoCs the abort is erroneously applied to the antecedent also. When simulating the checkers with the counterexample trace, in the FoCs checker the abort condition cancels the *hready* and the assertion does not trigger. In the MBAC checker the assertion signal triggers (as it should); this was confirmed in simulation where ModelSim's interpretation of the assertion on the counterexample trace also triggers.

For the MemSlave_ASR and ARB_ASR test cases, the equivalence verification succeeded with the monostable output FF only. As stated in the previous section,

**Table 7.1:** Benchmarking of non-synthetic assertions ( N.S.Y. = Not Supported Yet).

| | **Hardware Metrics** | | | | | | **Equivalence Check** |
|---|---|---|---|---|---|---|---|
| | MBAC | | | FoCs | | | |
| **Assertion** | FF | LUT | MHz | FF | LUT | MHz | MBAC $\leftrightarrow$ FoCs |
| PCI_ASR | 9 | 20 | 317 | 9 | **19** | 317 | pass |
| AHB_ASR | 18 | **17** | 611 | 18 | 18 | **665** | **fail** $\rightarrow$ CE$_1$ |
| MemSlave_ASR | 5 | 16 | **456** | 5 | **11** | 338 | pass |
| OVL_ASRa | 4 | 5 | 606 | 4 | 5 | 606 | pass |
| OVL_ASRb | 4 | 4 | 474 | 4 | **3** | 474 | pass |
| OVL_ASRc | 10 | **8** | 311 | 10 | 9 | 311 | pass |
| OVL_ASRd | 11 | 9 | **332** | 11 | **8** | 311 | pass |
| NOC_ASR | **3** | **69** | 559 | | N.S.Y. | | – |
| DATARX_ASRa | 5 | 5 | 598 | 5 | 5 | 598 | pass |
| DATARX_ASRb | 6 | 7 | 610 | 6 | 7 | 610 | pass |
| DATARX_ASRc | 4 | 11 | 667 | 4 | 11 | 667 | pass |
| DATARX_ASRd | 5 | **5** | **559** | 5 | 6 | 441 | pass |
| DATARX_ASRe | **13** | **21** | **439** | | N.S.Y. | | – |
| UDCNT_ASRa | 10 | 7 | 348 | 10 | 7 | 348 | pass |
| UDCNT_ASRb | 10 | 6 | 349 | 10 | 6 | 349 | pass |
| UDCNT_ASRc | 10 | **6** | **366** | 10 | 7 | 348 | pass |
| UDCNT_ASRd | 10 | 10 | 667 | 10 | 10 | 667 | pass |
| CPU_ASR | 6 | 4 | 564 | 6 | 4 | 564 | pass |
| ARB_ASR | 7 | 9 | 434 | 7 | 9 | 434 | pass |

the monostable does not affect the validity of the equivalence verification, and only indicates that the checkers have differences in their run-time semantics after the first error is reported. The difference in behavior explains the difference in the LUT metric in the table for this example.

The NOC_ASR and DATARX_ASRe test cases are not supported by FoCs. In the first case, FoCs does not support the use of eventually! in the consequent of the suffix implication, and in the second case it is likely that the strong sequence is the cause. In the second case no output is generated by FoCs even though the assertion is properly recognized in its user interface. The hardware metrics for the NOC_ASR assertion are heavily biased towards LUTs because of the many vector comparisons in the sequences. In the remaining test cases in the table, both tools perform very similarly in terms of circuit sizes, and are shown to be functionally equivalent.

## 7.3 Evaluating Assertion Grouping

In this section the assertion partitioning algorithm presented in Section 6.5 is evaluated. The MBAC checker generator is used to produce assertion checkers for two suites of assertions. The assertions are used to verify an AMBA slave device and AMBA AHB interface compliance, and are from Chapter 8 in “Using PSL/ Sugar for Formal and Dynamic Verification” [53]. Because of the temporal nature of the assertions, the checkers utilize more combinational cells than flip-flops. However, the partitioning algorithm can operate on any type of circuits whether they are balanced or biased towards either flip-flops or combinational logic.

Table 7.2 shows the individual resource usage of checkers for the assertions in the AHB and MemSlave examples. In the table, N.A. means Not Applicable, and occurs for circuits containing only one FF with no feedback path (the MHz is a clk-to-clk figure). Table 7.3 shows how the assertion circuits from Table 7.2 are partitioned into a minimal number of sets by the subset-circuit algorithm (Algorithm 6.2), for a target area of 50 FFs and 50 four-input LUTs. In both cases, results of phase two in the algorithm were logged (dominant LUTs). The right-most column lists the sums of the circuit metrics in each group.

Table 7.4 shows how the actual resource usage can be slightly diminished when the circuits that form a subset are actually synthesized together. As a general result, it can be expected that as the number of circuits per subset increases, the resource sharing becomes more important, and the overall metrics for a given subset become smaller. For comparison purposes, Table 7.4 also lists the full-set metrics, which are obtained by synthesizing all checkers as a single module. In both cases resource sharing is greater for LUTs than FFs, and is more prominent in the MemSlave case.

The end result is an efficient partition of checkers which minimizes the number of times the reprogrammable logic area must be reconfigured. A test procedure can then run a batch of test sequences with a given subset of checkers, then instantiate a new set of checkers, re-run the test sequences, and so forth. Once the verification with checkers is finished, the reprogrammable fabric can be used for the functionality of the intended design.

## 7.4 Pre-Synthesis Results

In this section the effects of the alphabet choice on the automata produced are evaluated. The automata metrics that are reported are the number of edges and the

**Table 7.2:** Resource usage of assertion checkers for grouping.

AHB example:

| Assertion | FFs | LUTs | MHz | Assertion | FFs | LUTs | MHz |
|---|---|---|---|---|---|---|---|
| AHB_A1 | 2 | 2 | 667 | AHB_A14 | 1 | 12 | N.A. |
| AHB_A2 | 2 | 3 | 611 | AHB_A15 | 1 | 36 | N.A. |
| AHB_A3 | 2 | 3 | 667 | AHB_A16 | 2 | 20 | 667 |
| AHB_A4 | 2 | 2 | 611 | AHB_A17 | 2 | 2 | 611 |
| AHB_A5 | 2 | 2 | 667 | AHB_A18 | 3 | 2 | 667 |
| AHB_A6 | 2 | 2 | 667 | AHB_A19 | 3 | 3 | 667 |
| AHB_A7 | 2 | 2 | 667 | AHB_A20 | 3 | 2 | 667 |
| AHB_A8 | 2 | 2 | 667 | AHB_A21 | 1 | 23 | N.A. |
| AHB_A9 | 2 | 2 | 667 | AHB_A22 | 1 | 21 | N.A. |
| AHB_A10 | 1 | 6 | N.A. | AHB_A23 | 1 | 21 | N.A. |
| AHB_A11 | 2 | 30 | 667 | AHB_A24 | 1 | 19 | N.A. |
| AHB_A12 | 2 | 18 | 667 | AHB_A25 | 1 | 6 | N.A. |
| AHB_A13 | 2 | 18 | 611 | AHB_A26 | 18 | 17 | 611 |

MemSlave example:

| Assertion | FFs | LUTs | MHz | Assertion | FFs | LUTs | MHz |
|---|---|---|---|---|---|---|---|
| MemSlave_A1 | 1 | 4 | N.A. | MemSlave_A14 | 2 | 3 | 667 |
| MemSlave_A2 | 2 | 4 | 667 | MemSlave_A15 | 2 | 3 | 667 |
| MemSlave_A3 | 2 | 2 | 667 | MemSlave_A16 | 2 | 7 | 667 |
| MemSlave_A4 | 2 | 2 | 667 | MemSlave_A17 | 1 | 2 | N.A. |
| MemSlave_A5 | 1 | 2 | N.A. | MemSlave_A18 | 4 | 12 | 442 |
| MemSlave_A6 | 1 | 7 | N.A. | MemSlave_A19 | 1 | 5 | N.A. |
| MemSlave_A7 | 1 | 2 | N.A. | MemSlave_A20 | 1 | 6 | N.A. |
| MemSlave_A8 | 1 | 7 | N.A. | MemSlave_A21 | 1 | 6 | N.A. |
| MemSlave_A9 | 4 | 9 | 417 | MemSlave_A22 | 1 | 1 | N.A. |
| MemSlave_A10 | 5 | 16 | 456 | MemSlave_A23 | 2 | 5 | 667 |
| MemSlave_A11 | 5 | 22 | 469 | MemSlave_A24 | 1 | 4 | N.A. |
| MemSlave_A12 | 2 | 3 | 667 | MemSlave_A25 | 1 | 3 | N.A. |
| MemSlave_A13 | 2 | 3 | 667 | MemSlave_A26 | 1 | 18 | N.A. |

**Table 7.3:** Checker partitions for reprogrammable area.

AHB example:

| **Subset** | **Assertion circuits in partition** | **ΣFF, ΣLUT** |
|---|---|---|
| #1 | {A9, A14, A15} | 4, 50 |
| #2 | {A8, A22, A23, A25} | 5, 50 |
| #3 | {A7, A10, A21, A24} | 5, 50 |
| #4 | {A6, A11, A13} | 6, 50 |
| #5 | {A1, A2, A3, A4, A5, A12, A16} | 14, 50 |
| #6 | {A17, A18, A19, A20, A26} | 29, 26 |
| | Total: | 63, 276 |

MemSlave example:

| **Subset** | **Assertion circuits in partition** | **ΣFF, ΣLUT** |
|---|---|---|
| #1 | {A6, A8, A19, A20, A21, A22, A26} | 7, 50 |
| #2 | {A1, A11, A15, A18, A23, A24} | 15, 50 |
| #3 | {A2, A3, A5, A7, A9, A10, A14, A16, A17, A25} | 21, 50 |
| #4 | {A4, A12, A13} | 6, 8 |
| | Total: | 49, 158 |

**Table 7.4:** Subset and full-set synthesis of checker groups.

AHB example:

| **Subset** | **FFs, LUTs** |
|---|---|
| #1 | 4, 50 |
| #2 | 5, 50 |
| #3 | 5, 49 |
| #4 | 6, 34 |
| #5 | 13, 48 |
| #6 | 29, 26 |
| Total: | 62, 257 |

| **Full-Set (FFs, LUTs)** |
|---|
| 60, 250 |

MemSlave example:

| **Subset** | **FFs, LUTs** |
|---|---|
| #1 | 7, 43 |
| #2 | 15, 47 |
| #3 | 21, 47 |
| #4 | 6, 8 |
| Total: | 49, 145 |

| **Full-Set (FFs, LUTs)** |
|---|
| 48, 129 |

number of states. In the tables to follow, the two columns titled "Edges $\Sigma_{\mathcal{P}}$" and "Edges $\Sigma_S$" represent the number of edges when either the power set or symbolic approaches are used, as defined in Definitions 4.1 and 4.2 respectively. Since the number of states is identical in both approaches, the number of states is reported in one column. After having considered Example 4.1, the power set alphabet can be expected to be worse; however, this section presents broader quantitative data to confirm this.

The goal of this section is two-fold: first, the efficiency of the two alphabet approaches is compared, and second, an example of how assertions and a checker generator can be used to create hardware pattern matchers is shown. Although this second case is illustrated using hardware protein matchers as examples, the purpose is not to compare with existing protein matching machines, whether hardware or software, but rather to show that assertions and a checker generator can be used to perform other types of automated circuit design, going beyond the intended task of verification.

### 7.4.1 Experiments with Hardware Protein Matchers

The string matching theme is explored here, with a particular application in protein matching. The checker generator can be used to generate protein matching circuits that could potentially perform the matching faster than software approaches [138] and require less computing infrastructure. The protein sequences that are to be matched are described by regular expressions in a slightly different notation than conventional regular expressions: "–" represents concatenation, "x" represents any symbol, symbols inside "[ ]" denote choice and parentheses represent repetition (with a count or a range). The proteins used in the experiments are from the PROSITE list [99].

The typical task of protein matching software consists in finding a given protein in a longer protein. A protein is a sequence of amino acids, and there are twenty standard amino acids each represented by a single capitalized character. For example, here is the expression for Protein Site (PS) # 00007:

$$\text{PS00007:}\quad [\text{RK}] - \text{x}(2) - [\text{DE}] - \text{x}(3) - \text{Y}$$

A pattern matcher for this regular expression can be expressed in PSL as the assertion:

$$\textsf{assert never } \{\{t\text{=='R'}\}|\{t\text{=='K'}\};[*2];\ \{t\text{=='D'}\}|\{t\text{=='E'}\};[*3];t\text{=='Y'}\}; \qquad (7.1)$$

The characters in quotes represent ASCII values for the corresponding protein sym-

**Table 7.5:** Hardware protein matching automata generated by MBAC.

| **Site #** | **Protein** | **# Edges** $\Sigma_S$ | **# Edges** $\Sigma_{\mathcal{P}}$ | **# States** |
|---|---|---|---|---|
| PS00007: | [RK]–x(2)–[DE]–x(3)–Y | 9 | 60 | 9 |
| PS00112: | C–P–x(0,1)–[ST]–N–[ILV]–G–T | 10 | 768 | 9 |
| PS00259: | Y–x(0,1)–[GD]–[WH]–M–[DR]–F | 9 | 352 | 8 |
| PS00328: | H–R–H–R–G–H–x(2)–[DE](7) | 16 | 152 | 16 |
| PS00354: | [AT]–x(1,2)–[RK](2)–[GP]–R–G–R–P–[RK]–x | 14 | 576 | 13 |
| PS00831: | G–x–[LIVM](2)–x–R–Q–R–G–x(5)–G | 16 | 192 | 16 |
| PS01088: | [LIVM](2)–x–R–L–[DE]–x(4)–R–L–E | 14 | 320 | 14 |

bols. The assertion in (7.1) postulates that the protein sequence is never matched. When it is matched, the assertion output signal triggers and indicates when the string is detected. Normally, an assertion failure indicates a problem in the device being verified, but in this case the checker is used as a pattern matcher that must report occurrences of a given sequence. Every time the checker output triggers, the pattern has been detected.

It should be noted that the circuit–under–verification need not be completely defined for the checker generator to operate correctly: all that must be done is to bind the assertion to a module that defines an 8–bit signal called $t$. The actual circuit would be a text reader that scans one character per clock cycle. The never property has the effect of continually starting a match for the sequence, such that every time the sequence manifests itself, the assertions output triggers.

Table 7.5 shows the automata metrics of the hardware protein matchers generated by MBAC, for a selection of protein sites. Each protein is expressed as an assertion in a similar manner to PS00007 shown in (7.1). As can be observed, the number of edges is noticeably smaller in the symbolic encoding. The hardware string (protein) matching created here runs in $O(m)$ time, where $m$ is the number of characters (amino acids) to be searched.

Hardware accelerated protein matching could be used for efficiently searching large protein databases (for example, the TIGR Microbial Database), and could also be adapted to DNA sequence matching, given that a DNA sequence uses only four types of amino acids. Whether for string or protein matching, and even network intrusion detection, the general area of hardware accelerated pattern matching is related to the checker generators used in assertion-based verifications through sequences.

### 7.4.2 Complex Sequences

For high performance applications, the technique used in the previous subsection to create a hardware circuit for pattern matching would represent a first step, and could be improved by such measures as pipelining and further parallelism. Handling more than one character per clock is a common solution to improve throughput. The previous experiment nonetheless shows the scope of how assertions and a checker generator can be used to create other kinds of hardware, not only checker hardware for assertion verification. In this section complex sequences are evaluated, and the effect of symbol alphabet choice is compared.

As observed in Table 7.2 in the previous section, typical assertions such as most of the assertions used for verifying bus protocols span few clock cycles and do not showcase the strength the checker generator because they are easily handled. In the remainder of this chapter, synthetic assertions designed to benchmark complex sequences and properties are employed. Because the Boolean layer does not add to the temporal complexity of assertion automata, without loss of generality the Boolean layer is abstracted away using simple signal names $a$, $b$, etc. As assertions become more popular and verification engineers become more adept at writing them, checker generators must be able to handle all language features and scale efficiently for temporally complex assertions.

Table 7.6 shows a set of test assertions used to evaluate the pre-synthesis metrics of the checkers produced by the MBAC and FoCs tools. The automata metrics for this set of properties, as implemented in RTL code, are presented in Table 7.7. The number of states is inferred by looking at the width of the register vector used to encode a given automaton in Verilog. In the other sections in this chapter, hardware metrics are compared after each checker is synthesized for FPGA technology. In this section, results are presented that more closely relate the size of the automata that are actually generated, before any hardware synthesis optimizations.

The number of edges using both alphabet approaches in MBAC is shown in the second and third columns. As can be observed, the symbolic alphabet produces a much simpler automaton. In both cases the number of states is identical, as reported in the fourth column. The number of states is reported for the FoCs checker generator in the last column. As stated in the literature, FoCs is also automata-based [58, 106]. A comparison to FoCs was not attempted for the number of edges, given that their automata-to-RTL encoding is not documented, and inaccurate data could be inferred. From the results, it can be observed that the automata produced by MBAC are more compact than those produced by FoCs, and in some cases by more than an order of

**Table 7.6:** Pre-synthesis benchmarking properties.

| **Assertion** “assert P$x$;”, where P$x$ is: | |
|---|---|
| P1 | never {{a[*0:1] ; b[*0:2]} : {c[*0:1] ; d}} |
| P2 | never {{a[*] ; b[*1:3]} \| {c ; d[*1:2] ; e}} |
| P3 | never {{a[*]} : {b[*]}} |
| P4 | never {{a \| b} ; {{c[*]} && {d[*1:3]}} : {e}} |
| P5 | never {{a ; b[*]} : {c[*] ; d} ; e} |
| P6 | never {{{b ; c[*2:4] ; d}[+]} && {b ; {e[->2:4]} ; d}} |
| P7 | never {{a ; b[*1:3] ; c[*0:1]} & {d[*2:4] ; e[->]}} |
| P8 | never {{a[*0:1] ; b[*1:2] ; c[*]} : {d[*0:1] ; e[*2:4]}} |
| P9 | never {{a ; b[*] ; c[*]} : {d[*] ; e[*]} ; f[*]} |
| P10 | never {{a ; b[*] ; c[*]} : {d[*] ; e[*2:4]} ; f[->]} |
| P11 | never {{a[*] ; b[*] ; c[*]} && {d[*5:7]}} |
| P12 | never {{{a[*] ; b[*] ; c[*]} && {d[*5:7]}} : {c[->]}} |
| P13 | always {a} \|=> {{{c[*1:3] ; d}[+]} && {{e[->2:3]} ; d}} |
| P14 | always {a} \|=> {{{b ; c[*1:2] ; d}[+]} : {{e[->]} ; d}} |
| P15 | always {a} \|=> {{{b ; c[*1:2] ; d}[+]} : {b ; {e[->2:3]} ; d}} |
| P16 | always {a} \|=> {b ; {{c[*0:2]} ; {d[*0:2]}}[*] ; e} |
| P17 | always {a} \|=> {b ; {c ; d}[*] ; e ; {f ; b}[+] ; g} |
| P18 | always {a} \|=> {{b ; c[*]} : {d[*] ; e} ; f} |

magnitude. For test cases P13 to P15, no output was produced by FoCs.

The actual run times required to compile the assertions are also interesting to compare. To compile the most demanding case fifty times (P13 in the table), FoCs requires approximately 22 seconds, while MBAC requires less than 1/8 of a second. An intermediate test case is P12, where FoCs requires approximately 2 seconds and MBAC takes less than 1/10 of a second, where the assertion is also compiled fifty times. These results were obtained on a 2.4 GHz Pentium 4 with 512MB RAM.

## 7.5 Benchmarking Debug Enhancements

The effects of assertion threading, assertion completion and activity monitors are explored in this section by synthesizing the assertion circuits produced by the checker generator. The signal-dependency logging that was presented in Subsection 6.4.1 does not influence the circuits generated by the checker generator, while the assertion and coverage counters from Subsection 6.4.4 contribute a hardware overhead that is easily determined *a priori* (i.e. $n$-bit counter with saturation logic).

Some of the assertions used in this section are from the Cohen [53] and Foster [74]

**Table 7.7:** Pre-synthesis benchmark results for checkers generated by MBAC and FoCs (N.O. = No Output).

| | **MBAC** | | | **FoCs** |
|---|---|---|---|---|
| **Assertion** | Edges $\Sigma_S$ | Edges $\Sigma_{\mathcal{P}}$ | States | States |
| P1 | 6 | 40 | **3** | 19 |
| P2 | 7 | 64 | **5** | 16 |
| P3 | 3 | 7 | **2** | 6 |
| P4 | 7 | 50 | **5** | 10 |
| P5 | 10 | 144 | **5** | 12 |
| P6 | 68 | 324 | **27** | 98 |
| P7 | 26 | 320 | **11** | 74 |
| P8 | 19 | 112 | **7** | 101 |
| P9 | 27 | 656 | **6** | 69 |
| P10 | 33 | 800 | **9** | 44 |
| P11 | 28 | 124 | **14** | 109 |
| P12 | 33 | 140 | **15** | 736 |
| P13 | 90 | 296 | **21** | N.O. |
| P14 | 114 | 624 | **23** | N.O. |
| P15 | 221 | 760 | **66** | N.O. |
| P16 | 7 | 96 | **4** | 8 |
| P17 | 32 | 1216 | **11** | 15 |
| P18 | 42 | 480 | **10** | 18 |

books, while others were created during the development of MBAC to exercise the checker generator as thoroughly as possible. In the AMBA, PCI and CPU example assertions appearing in the tables in this section, complex Boolean layer expressions are replaced by simplified Boolean symbols. Synthesizing the full expressions does not change the temporal complexity of the automata because the Boolean layer expressions are not incorporated directly into assertion automata, but rather instantiated separately in HDL, where a result signal is used. Synthesizing assertions with simple Boolean expressions actually emphasizes the logic required for capturing the temporal structure of an assertion, and is the preferred method.

Table 7.8 shows a set of test assertions used to benchmark the debugging enhancements that were introduced in Section 6.4. The assertions checkers are synthesized in the normal operating mode of the checker generator; these results are shown in the first column of triples in the top half of Table 7.9.

As described in Section 6.4.3, assertions can alternately be compiled in completion mode as opposed to the typical failure mode. Synthesis results for the completion mode checkers are shown in the second group of triples in the top half of Table 7.9.

**Table 7.8:** Test assertions for debugging enhancements. (′ = Simplified Booleans.)

| Assertion | |
|---|---|
| A1 | assert always {a&b} \|-> {~c; {d&~c}[*0:4]; c&~d}; |
| A2 | assert always ({a} \|=> {{c[*0:1];d}\|{e}}); |
| A3 | assert always ({a;b} \|=> {c[*0:1];d}); |
| A4 | assert always {a} \|=> {{[*2];b;~c} \| {[*2];~b;c}}; (Example 6.4) ′ |
| A5 | assert always {a} \|=> {b;c;d;e}; (AMBA asr. [53]) ′ |
| A6 | assert always {a;~a} \|=> {(~a)[*0:15];a} abort b; (AMBA asr. [53]) ′ |
| A7 | assert always {a;b} \|=> {c;{{d[*];e}[+];f}&&{g[*]}} abort h; (PCI [74]) ′ |
| A8 | assert always {a} \|=> {e;d;{b;e}[*2:4];c;d}; |
| A9 | assert always {a} \|=> {b; {c[*0:2]} \| {d[*0:2]} ; e}; |
| A10 | assert always {a} \|=> {{{c[*1:2];d}[+]} && {{e[->2:3]};d}}; |
| A11 | assert always {a} \|=> {{{b;c[*1:2];d}[+]} & {b;{e[->2:3]};d}}; |
| A12 | assert always {a} \|=> {{{b;c[*1:2];d}[+]} && {b;{e[->2:3]};d}}; |
| A13 | assert never {a;d;{b;a}[*2:4];c;d}; |
| A14 | assert never { {{b;c[*1:2];d}[+]} && {b;{e[->2:3]};d} }; |
| A15 | assert always {a} \|=> {{{b;c[*1:2];d}[+]} : {b;{e[->]};d}}; |

From the table, it can be observed that a completion-mode assertion utilizes slightly less combinational logic (LUTs), and on average runs slightly faster than its normal-mode version (i.e. regular failure matching). Test assertions A13 and A14 have exactly the same metrics as in the normal mode since the completion mode has no effect on those types of properties (assert never *seq*).

The activity monitors introduced in Section 6.4.2 are used to observe when sequences are undertaking a matching. The third set of triples in the top half of Table 7.9 shows the hardware metrics of the checkers with activity monitors, for the example assertions. Unless specified, the default mode is not completion mode but rather the typical failure mode. Experiments could also be performed for the combination of activity monitors with completion mode. As can be noticed in the table, the maximum operating frequency is slightly diminished compared to the normal checkers, and in some cases, an additional flip-flop is required. The effect of the OR-gate appearing in the state-signal disjunction that is used to form the activity signal is visible in the slight increase in the LUT metric.

The checkers produced in the assertion threading experiments, which are reported in the bottom half of the table, were formally verified by model checking and were proven to be equivalent to their normal counterparts in the top-left column in the table. All equivalence verifications succeeded with the exceptions of the 8-way thread-

**Table 7.9:** Resource usage of checkers with debugging enhancements.

| | **Normal** | | | **Completion** | | | **Activity** | | |
|---|---|---|---|---|---|---|---|---|---|
| **Assertion** | FF | LUT | MHz | FF | LUT | MHz | FF | LUT | MHz |
| A1 | 6 | 8 | 433 | 6 | 7 | 444 | 6 | 11 | 429 |
| A2 | 3 | 3 | 610 | 3 | 2 | 610 | 3 | 4 | 610 |
| A3 | 4 | 3 | 611 | 4 | 3 | 553 | 4 | 5 | 564 |
| A4 | 6 | 3 | 564 | 6 | 3 | 564 | 6 | 5 | 559 |
| A5 | 5 | 5 | 514 | 5 | 4 | 611 | 5 | 6 | 509 |
| A6 | 18 | 17 | 611 | 18 | 17 | 502 | 18 | 23 | 564 |
| A7 | 5 | 10 | 468 | 5 | 7 | 470 | 5 | 12 | 411 |
| A8 | 15 | 21 | 329 | 15 | 15 | 430 | 15 | 26 | 312 |
| A9 | 7 | 12 | 333 | 7 | 8 | 412 | 7 | 14 | 331 |
| A10 | 16 | 38 | 304 | 16 | 31 | 386 | 17 | 40 | 293 |
| A11 | 44 | 141 | 260 | 44 | 139 | 250 | 44 | 150 | 259 |
| A12 | 35 | 118 | 251 | 35 | 100 | 281 | 35 | 128 | 243 |
| A13 | 12 | 11 | 559 | 12 | 11 | 559 | 12 | 15 | 559 |
| A14 | 12 | 12 | 456 | 12 | 12 | 456 | 13 | 17 | 452 |
| A15 | 26 | 80 | 246 | 26 | 79 | 250 | 27 | 87 | 249 |

| | **Assertion Threading** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **2-way** | | | **4-way** | | | **8-way** | | |
| | FF | LUT | MHz | FF | LUT | MHz | FF | LUT | MHz |
| A1 | 15 | 18 | 386 | 29 | 33 | 306 | 57 | 62 | 241 |
| A4 | 15 | 11 | 442 | 29 | 20 | 362 | not required | | |
| A5 | 13 | 16 | 323 | 25 | 24 | 326 | not required | | |
| A6 | 39 | 38 | 442 | 77 | 75 | 364 | 153 | 144 | 297 |
| A7 | 13 | 23 | 297 | 25 | 39 | 287 | 49 | 67 | 235 |
| A8 | 33 | 44 | 298 | 65 | 83 | 252 | 129 | 164 | 235 |
| A12 | 73 | 235 | 239 | 145 | 430 | 213 | 289 | 900 | 186 |
| A13 | 25 | 23 | 564 | 49 | 46 | 433 | 97 | 91 | 408 |
| A14 | 29 | 35 | 410 | 57 | 70 | 410 | 113 | 139 | 408 |
| A15 | 57 | 165 | 252 | 113 | 301 | 205 | 225 | 570 | 177 |

ing of A8, the 4-way and 8-way threading of A15 and all versions of A12, which exceeded the maximum memory capacity of the model checker. However, in these exceptions, no counter-example was produced during the run time of the tool. The verification performed here shows that threading preserves the intended functionality of the checker circuit, while increasing the observability in the checking process. The equivalence check also confirms that the dispatcher circuit and the overall threading strategy is correct, on the examples tested.

As presented in Section 6.4.5, assertion threading replicates sequence circuits in order for the failure conditions to be isolated from other activations. This was shown to ease the debugging process considerably, particularly when temporally complex assertions are used. The lower half of Table 7.9 shows how the resource utilization scales as a function of the number of hardware threads. Because 8-way threading is only useful for sequences that span at least eight clock cycles, the assertions used must have a certain amount of temporal complexity for the results to be meaningful. Since the assertion derived from Example 6.4 in A4 and the AMBA assertion in A5 both contain simple left-sides for the suffix implication, along with right-side sequences that span four clock cycles, these assertions do not benefit from 8-way assertion threading.

In the current version of the checker generator, when a sequence is a simple sequence such as $\{a\}$, it is still threaded even though doing this is useless. The threading results in the table could be improved if this was detected, and such simple sequences were not threaded. As anticipated, the experimental data shows that the resource utilization scales linearly with the number of hardware threads.

Test case A7 corresponds to the PCI_ASR test case from Section 7.2 with simple Booleans. The full assertion was shown to require 9 FFs and 20 LUTs, as opposed to 5 FFs and 10 LUTs in Table 7.9. The full PCI assertion exhibits particularly large overhead compared to the simplified one, as some intermediate variables are sequential (e.g., *cbe_stable*).

The 4-way threaded assertion circuit as used in the CPU execution example (Example 6.4) with non-simplified Boolean expressions actually synthesizes to 29 FFs and 21 LUTs, with a maximum frequency of 362 MHz. This corresponds to virtually the same metrics as the simplified version used in test case A4. The non-threaded checker for this test case required only 1 less LUT, compared to the non-simplified version in Section 7.2. Test cases A5 and A6 are based on the MemSlave_ASR and AHB_ASR test cases from Section 7.2, with simplified Booleans. In the simplified cases, the number of LUTs was reduced from 16 to 5 for the MemSlave_ASR checker and remained identical for the AHB_ASR checker.

**Table 7.10:** Benchmarking implementations of eventually!.

| **Property** (F ≡ eventually!, G ≡ always) | **Splitting** | | | **Rewrite** | | |
|---|---|---|---|---|---|---|
| | FF | LUT | MHz | FF | LUT | MHz |
| eventually! {b;c;d} | **4** | **4** | **559** | 5 | 11 | 388 |
| eventually! {a;b;c;d;e} | **6** | **6** | **559** | 17 | 52 | 237 |
| always (a −> eventually! {b;c;d}) | **4** | **4** | **559** | 5 | 10 | 395 |
| G (a−> eventually! {a;b;c;d;e}) | **6** | **5** | **564** | 17 | 44 | 236 |
| G (a−> eventually! {b[*5:10]}) | 6 | 6 | **559** | 6 | **5** | 548 |
| G (a−> F{b; {c[*0:2]} \| {d[*0:2]} ; e}) | **6** | **8** | **444** | 7 | 20 | 329 |
| G (a−> F{ {{c[*1:2];d}[+]} \| {e[−>2]} }) | 6 | **8** | **434** | **5** | 9 | 392 |
| G (a−> F{{{c[*1:2];d}[+]}:{{e[−>]};d}}}) | 6 | 7 | **417** | **5** | 7 | 395 |
| G (a−> F{{{c;d}[*]}&&{∼e[*];e;∼e[*]}}}) | 4 | **5** | **474** | 4 | 7 | 465 |
| G ({a;b[*0:2];c} \|−> F {d;e[*1:3];f}) | **9** | **9** | **473** | 10 | 17 | 309 |
| G (a−> F { {{b;c[*1:2];d}[+]} && {b;{e[−>2:3]};d} }) | **17** | **27** | **395** | 68 | 245 | 207 |

## 7.6 Benchmarking Sequences and Properties

In this section the two eventually! approaches, namely the rewrite rule and the automata splitting technique, are compared. The assertion circuits produced by the MBAC checker generator are also compared to those produced by FoCs for complex sequences and properties. In the experiments, simple Boolean expressions are used because the emphasis is placed on the temporal behavior of the checkers. Ideally, the assertion circuits that are produced should be small, fast, should support all simple subset operators and should exhibit the correct behavior.

### 7.6.1 Comparison of the Two eventually! Approaches

Table 7.10 shows the advantages of the split-automata method in the implementation of eventually!, as presented in Section 6.3, compared to the rewrite rule presented in Section 6.3. The split-automata method scales much better because a typical matching automaton can be employed as opposed to a failure matching automaton, which can be exponentially larger given the required strong determinization. In the test cases, the split-automata method produces faster circuits, and except for a few small examples, requires less hardware. In all eleven test cases, functional equivalence of the checkers was formally verified by model checking. These examples show that in general not all sequential optimizations can be performed by traditional synthesis tools, and efforts to optimize the checkers pre-synthesis should always be made.

**Table 7.11:** Occurrence-matching test sequences.

| **Assertions** “assert never S$x$;”, where S$x$ is | |
|---|---|
| S1 | { {a[∗];b[∗1:3]} \| {c;d[∗1:2];e} } |
| S2 | { {a\|b} ; {{c[∗]} && {d[∗1:3]}} : {e} } |
| S3 | { {a;[∗];b} && {c[∗1:5];d} } |
| S4 | { {a\|b};{{c[∗]} && {d[∗1:3]}}:{e} } |
| S5 | { {a\|b};{{c[∗]} && {d[∗1:6]}}:{e} } |
| S6 | { a ; {b ; c}[∗] ; d ; {e ; a}[+] ; f } |
| S7 | { {e;e} within {c;d;a;b;c} } |
| S8 | { {a;b[∗1:3]} & {c[∗2:4]} } |
| S9 | { {a;b[∗1:3];c[∗0:1]} & {d[∗2:4];e[->]} } |
| S10 | { {{b;c[∗1:2];d}[+]} && {b;{e[->2:3]};d} } |
| S11 | { {{b;c[∗2:4];d}[+]} && {b;{e[->2:4]};d} } |
| S12 | { {a ; b[∗]} : {c[∗] ; d} ; e } |
| S13 | { {a ; b[∗] ; c[∗]} : {d[∗] ; e[∗]} ; f[∗] } |
| S14 | { {a ; b[∗] ; c[∗]} : {d[∗] ; e[∗2:4]} ; f[->] } |
| S15 | { {a[∗0:1];b[∗0:2]} : {c[∗0:1];d} } |
| S16 | { {a[∗0:1];b[∗1:2];c[∗]} : {d[∗0:1];e[∗2:4]} } |
| S17 | { {a[∗];b[∗];c[∗]} && {d[∗5:7]} } |
| S18 | { {{a[∗];b[∗];c[∗]} && {d[∗5:7]}} : {c[->]} } |

## 7.6.2 Occurrence-Matching Sequences

The FoCs and MBAC checker generators are evaluated with the set of assertions shown in Table 7.11, and the results appear in Table 7.12. These results illustrate the efficiency of the implementation of occurrence-matching SEREs, with partial emphasis on intersection and fusion operators. In the comparison table, N.A. (Not Applicable) appears when an assertion circuit contains only one FF and the FF has no feedback path (the MHz is a clk-to-clk figure). In all cases, the circuits produced by MBAC are smaller or equal in size, and in all but two cases have an equal or higher operating frequency.

For test cases S15, S16 and S18, the FoCs and MBAC checkers do not have the same behavior. The counterexamples reported by the model checker show that immediately upon releasing the reset, when the sequences

CE$_2$: $\{ \sim reset \ ; \ b \wedge d \ \}$

CE$_3$: $\{ \sim reset \ ; \ b \wedge e \ ; \ e \ \}$

CE$_4$: $\{ \sim reset \ ; \ a \wedge d \ ; \ d \ ; \ d \ ; \ d \ ; \ c \wedge d \ \}$

execute (signals not listed are assumed false), the assertion signals from FoCs fail to

**Table 7.12:** Benchmarking of occurrence-matching sequences.

| | **Hardware Metrics** | | | | | | **Equivalence Check** |
|---|---|---|---|---|---|---|---|
| | MBAC | | | FoCs | | | |
| **S**$x$ | FF | LUT | MHz | FF | LUT | MHz | MBAC $\leftrightarrow$ FoCs |
| S1 | 4 | 3 | 564 | 4 | 3 | 564 | pass |
| S2 | 4 | 3 | **665** | 4 | 4 | 610 | pass |
| S3 | 6 | 6 | **667** | 6 | 6 | 559 | pass |
| S4 | 4 | **3** | **665** | 4 | 4 | 610 | pass |
| S5 | 7 | **6** | **665** | 7 | 8 | 444 | pass |
| S6 | 6 | 6 | 564 | 6 | 6 | 564 | pass |
| S7 | 11 | 10 | 564 | 11 | 10 | 564 | pass |
| S8 | **5** | **6** | **504** | 6 | 10 | 332 | pass |
| S9 | 10 | **16** | 325 | 10 | 21 | **331** | pass |
| S10 | **12** | **12** | **456** | 14 | 16 | 429 | pass |
| S11 | **20** | **21** | **456** | 32 | 39 | 383 | pass |
| S12 | **4** | **5** | 418 | 5 | 6 | **469** | pass |
| S13 | **5** | **7** | **352** | 32 | 56 | 261 | pass |
| S14 | **8** | **15** | **338** | 26 | 32 | 332 | pass |
| S15 | **2** | **2** | **665** | 6 | 7 | 395 | **fail** $\rightarrow$ CE$_2$ FoCs |
| S16 | **6** | **7** | **392** | 38 | 43 | 340 | **fail** $\rightarrow$ CE$_3$ FoCs |
| S17 | **13** | **15** | **506** | 19 | 33 | 412 | pass |
| S18 | **14** | **16** | **445** | 65 | 127 | 257 | **fail** $\rightarrow$ CE$_4$ FoCs |

report the error. In these cases, the sequences exhibit a violation of the respective properties, and must be detected as such by the assertion checker circuits. The reset signal is active low and is declared as an input to the checkers generated by both tools.

### 7.6.3 Failure-Matching Sequences

Both tools are evaluated using sequences that require failure matching, using the assertions shown in Table 7.13. Synthesis results are presented in Table 7.14. In the comparison table, "N.O." (No Output) denotes the cases when FoCs was not able to generate checkers for given assertions and no code was produced (5 of 18 cases).

In more than half of the test cases, large differences in checker behavior were observed between both tools. In such cases the confidence in the correctness of the MBAC checkers is further increased by comparing checker outputs in simulation to ModelSim's interpretation of PSL.

The *trace distance* is introduced to compare the behavior of two assertion-circuit

**Table 7.13:** Failure-matching test sequences.

| **Assertions** | |
|---|---|
| “assert always {a} \|=> S$x$;”, where S$x$ is | |
| S1 | { b;c[∗];d } |
| S2 | { {b;c;d} & {e;d;b} } |
| S3 | { e;d;{b;e}[∗2:4];c;d } |
| S4 | { b ; {c[∗0:4]} & {d} ; e } |
| S5 | { b ; {c[∗0:6]} & {d} ; e } |
| S6 | { {{c;d}[+]} && {e[−>2]} } |
| S7 | { {{c;d}[+]} && {e[−>4]} } |
| S8 | { {{c;d}[+]} && {e[−>6]} } |
| S9 | { {{c[∗1:2]}[+]} && {e[−>2]} } |
| S10 | { {{c[∗1:2];d}[+]} && {e[−>2]} } |
| S11 | { {{c[∗1:3];d}[+]} && {{e[−>2:3]};d} } |
| S12 | { {{b;c[∗1:2];d}[+]} : {{e[−>]};d} } |
| S13 | { {{b;c[∗1:2];d}[+]} : {b;{e[−>2:3]};d} } |
| S14 | { b ; {{c[∗0:2]} ; {d[∗0:2]}}[∗] ; e } |
| S15 | { {{c[∗1:2];d}[+]} & {e[−>2]} } |
| S16 | { {e;e} within {c;d;a;b;c} } |
| S17 | { b ; {c ; d}[*] ; e ; {f ; b}[+] ; g } |
| S18 | { {b ; c[*]} : {d[*] ; e} ; f } |

outputs. For two given traces of assertion signals, the trace distance is defined as the number of clock cycles in which the two signals disagree. The comparison is done at the rising edge of the clock. The assertion signals in question are typically from two different implementations of the *same* PSL assertion. The discrepancies between MBAC and ModelSim are shown in the right-most column, by measuring the number of cycles in the assertion output traces for which the two differ. For the software metrics to be meaningful, the assertions are made to trigger often during a simulation run. To accomplish this, primary signals supplied by the testbench are pseudo-randomly generated with different probabilities.

Regarding the semantics of PSL in dynamic verification, it should be noted that some assertions interpreted by simulators such as ModelSim will only trigger once for any given start condition. For example, in the following assertion,

$$\textsf{assert never } \{a;b[*0{:}1]\};$$

for a given cycle in which $a$ is asserted, the assertion will trigger. However, if the cycle that follows $a$ has $b$ asserted, the assertion will not trigger. This is perfectly acceptable

**Table 7.14:** Benchmarking of failure-matching sequences (M.C. = proven equivalent in Model Checking).

| **S**$x$ | **Hardware Metrics** | | | | | | **Assertion Distances** | |
|---|---|---|---|---|---|---|---|---|
| | MBAC | | | FoCs | | | MBAC–FoCs | MBAC–MSim |
| | FF | LUT | MHz | FF | LUT | MHz | | |
| S1 | 3 | 4 | 514 | 3 | **3** | **610** | 3272 | 0 |
| S2 | 4 | **5** | **445** | 4 | 8 | 393 | 0 (M.C.) | 0 |
| S3 | 15 | **21** | **329** | 15 | 22 | 292 | 0 (M.C.) | 0 |
| S4 | 7 | **9** | **441** | 7 | 10 | 333 | 3973 | 0 |
| S5 | 9 | **13** | **395** | 9 | 15 | 283 | 3973 | 0 |
| S6 | **5** | **6** | **509** | 6 | 8 | 395 | 0 (M.C.) | 0 |
| S7 | **9** | **13** | **326** | 10 | 14 | 325 | 0 (M.C.) | 0 |
| S8 | **13** | **18** | 324 | 14 | 19 | 324 | 0 (M.C.) | 0 |
| S9 | **3** | **3** | **564** | 4 | 4 | 429 | 0 (M.C.) | 0 |
| S10 | **9** | **22** | **311** | No Output | | | – | 0 |
| S11 | **20** | **52** | **278** | No Output | | | – | 209 |
| S12 | **22** | **59** | **259** | No Output | | | – | 0 |
| S13 | **65** | **222** | **250** | No Output | | | – | 0 |
| S14 | **3** | **4** | **610** | 7 | 12 | 331 | 26 | 0 r2061 |
| S15 | **4** | **4** | **472** | No Output | | | – | 0 |
| S16 | **12** | **16** | **386** | 17 | 26 | 302 | 236 | 270 |
| S17 | **10** | **22** | **314** | 14 | 33 | 281 | 0 (M.C.) | 0 |
| S18 | **9** | **17** | **280** | 17 | 44 | 258 | 0 (M.C.) | 0 |

given that the run-time semantics of PSL is not specified. If an assertion fails at one or more time points, it has failed globally. One possible reason for ModelSim's behavior is that in the simulation kernel, the threads which monitor assertions are kept as short as possible for performance reasons. In hardware this is not a concern; in MBAC the assertion triggers when "$a$" is observed, *and* when "$a; b$" is observed. Because of this difference, measuring distances between MBAC and ModelSim is more involved.

Between MBAC's assertion circuits and ModelSim, the distance is not incremented when an assertion circuit output triggers *and* ModelSim's assertion does not. When such a condition occurs, a residual distance is instead incremented. Residual distances are an indication that MBAC is reporting more failures, which can then be exploited for debugging purposes. Residual distances are noted "r $n$". When applicable, the residual distance is well anticipated because of the multiple paths in the corresponding sequence. In all cases, $10^5$ pseudorandom test vectors are supplied by the testbench.

The random stimulus comparison to ModelSim is obviously not a proof that the circuits generated by MBAC are correct, however it does offer reasonable assurance given the length of test sequences. The eight test cases with a distance of zero to FoCs were formally verified in model checking and are equivalent to FoCs' checkers.

The two cases of non-zero distances to ModelSim (S11 and S16) occur because MBAC's circuits are able to identify certain failures *earlier* than ModelSim. This arises when the automata are able to reach a final state earlier when evaluating a given sequence. The strength of the checker generator becomes apparent when increasing the complexity of sequences. The dual-level symbol tables, and the particularities in the minimization function pertaining to `true` edges and nondeterminism are the driving factors behind the efficient checkers produced by the tool. In all experiments, checkers are generated instantly by MBAC.

## 7.6.4 Properties

The assertion circuits produced by the MBAC checker generator are also evaluated using various test cases involving a variety of property operators. The FoCs and MBAC checker generators are compared using the set of assertions shown in Table 7.15, and the synthesis results are reported in Table 7.16. Properties P18 and P20-P24 are from a TIMA publication [24]. For cases P1 to P12 and P16, no synthesis results are given because the properties are not supported by FoCs. Property P14 exceeded the internal limits in FoCs and no output was produced.

With the exception of P13, when FoCs is able to produce a checker both tools produce functionally equivalent checkers. Functional equivalence was formally verified using model checking. Test cases P20 and P24 exceeded the maximum memory capacity of the model checker, and were compared in simulation instead. The FoCs checkers for those two cases have 309 and 175 state elements respectively in the HDL code, versus 27 and 9 respectively in MBAC's checkers.

Test cases P20 and P24 were compared using a testbench of $10^5$ biased pseudo-random test vectors. For each assertion, the circuits produced by both tools exhibit the same behavior on every clock cycle. In biased random vector generation, signal probabilities are adjusted in order for the assertions to trigger reasonably often. This method is not a proof that the circuits are functionally equivalent; however, combined with the fact that model checking produced no counterexample before reaching its limit, this does offer reasonable assurance.

For test case P13, slight differences in behavior were noticed due to the unspecified run-time semantics of "$p$ `until` $b$", where it is up to the tool's architect to decide whether to flag all failures of $p$ before $b$ occurs, or to flag only the first one. This flexibility is

**Table 7.15:** Benchmarking of properties.

| **Property** (“assert P$x$;”, where P$x$ is:) | |
|---|---|
| P1 | always ({a;d} \|-> next_e[2:4](b)) until c |
| P2 | always ({a;b} \|-> eventually! {c;d}) abort e |
| P3 | always {a;b[*0:2];c} \|=> ({d[*2]} \|-> next ∼e) |
| P4 | always (a -> ( (eventually! b[*5]) abort c) ) abort d |
| P5 | always ( (a -> (b before c)) && (c -> eventually! {b;d}) ) abort e |
| P6 | always {a;b;c} \|=> never {d[*0:3];e} |
| P7 | always a -> next_a![2:4](b) |
| P8 | always (a -> {[*0:7];b}) abort ∼c |
| P9 | always a -> next_e![2:4](b) |
| P10 | always a -> next_event_e!(b)[2:4](c) |
| P11 | always a -> ({b;c} until! d) |
| P12 | always a -> next_event_a!(b)[5:10](c) |
| P13 | always a -> (b until! c) |
| P14 | always {a} \|-> {{b;c[*]}:{d[*];e;f}} |
| P15 | never {a;[*];{b;c}[+]} |
| P16 | always (e \|\| (a -> ({b;c} until d))) |
| P17 | always a -> (b before!_ c) |
| P18 | always (a -> next_event_e(b)[1:6](c)) |
| P19 | always a -> eventually! b |
| P20 | always (a -> next (next_a[2:10](next_event(b)[10]((next_e[1:5](d)) until (c))))) |
| P21 | always ((a -> next(next[10](next_event(b)((next_e[1:5](d)) until (c))))) \|\| e) |
| P22 | (always (a -> next(next[10](next_event(b)((next_e[1:5](d)) until (c)))))) && (always (e -> (next_event_a(f)[1:4](next((g before h) until (i)))))) |
| P23 | always (a -> (next_event_a(b)[1:4](next((d before e) until (c))))) |
| P24 | always (a -> (next_event(c)((next_event_e(d)[2:5](e)) until (b)))) |

expected in dynamic verification with PSL, and may occur with other operators.

With test cases P17 and P19, the behavior of the checkers is identical between both tools, and only a slight difference occurs when the End-Of-Execution (EOE) signal activates. This was circumvented by using a monostable flip-flop on the assertion outputs such that when the assertion triggers, the output remains in the triggered state. The monostable FF was used because the semantics of the checkers does not have to be defined after the EOE occurs. In all test cases where FoCs is able to produce a checker, the circuits produced by MBAC are more resource-efficient than those produced by FoCs.

A subset of the test cases contained in Section 7.6 was also evaluated with another

**Table 7.16:** Benchmarking of properties (P1-P12 are not supported by FoCs yet).

| **P**$x$ | MBAC | | | **P**$x$ | MBAC | | | FoCs | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | FF | LUT | MHz | | FF | LUT | MHz | FF | LUT | MHz |
| P1 | **6** | **5** | **611** | P13 | **2** | **3** | **612** | 3 | 4 | 474 |
| P2 | **4** | **6** | **470** | P14 | **8** | **23** | **295** | No Output | | |
| P3 | **7** | **6** | **611** | P15 | **3** | **2** | **611** | 4 | 3 | 564 |
| P4 | **6** | **9** | **460** | P16 | **3** | **5** | **469** | Not Supported Yet | | |
| P5 | **4** | **7** | **473** | P17 | **2** | **3** | **610** | 3 | 4 | 474 |
| P6 | **7** | **7** | **419** | P18 | **7** | **8** | 445 | 12 | 12 | **564** |
| P7 | **5** | **2** | **445** | P19 | 2 | 2 | 564 | 2 | 2 | 564 |
| P8 | **8** | **8** | **667** | P20 | **26** | **10** | **375** | 200 | 106 | 299 |
| P9 | **5** | **4** | **433** | P21 | **17** | **7** | 564 | 23 | 13 | 564 |
| P10 | **5** | **6** | **456** | P22 | **23** | **11** | **513** | 46 | 43 | 312 |
| P11 | **3** | **4** | **505** | P23 | **7** | **5** | **552** | 24 | 24 | 311 |
| P12 | **11** | **7** | **326** | P24 | **8** | **11** | **439** | 40 | 40 | 408 |

synthesis tool for added confidence in the results. Test cases for which the differences in metrics are the greatest between the MBAC and FoCs circuits were selected. The test cases in question are:

- The second, fourth, and eleventh (last) assertion in Table 7.10 (comparison of the two eventually! approaches);
- S11, S13, S14 and S17 in Table 7.11 (occurrence-matching sequences);
- S14, S16, S17 and S18 in Table 7.13 (failure-matching sequences);
- P18 and P20 to P24 in Table 7.15 (properties);

The checkers for the assertions mentioned above were further synthesized using Altera QuatusII, for a StratixII EP2S15F672C3 FPGA. Because the Xilinx and Altera technologies differ, the number of LUTs is not directly comparable; however, qualitatively the results are coherent with the Xilinx metrics. The number of flip-flops is directly comparable and in all but two cases was identical in both the Altera and Xilinx experiments. For FoCs' checkers, test cases S13 and S14 in Table 7.11 synthesize to 15 and 25 FFs respectively in the Altera synthesis tool, compared to 32 and 26 FFs in the Xilinx tool. In both cases though, the checkers produced by MBAC are still three times smaller.

# Chapter 8

# Conclusion and Future Work

## 8.1 Conclusion

As assertion-based verification and emulation become increasingly important in verification, a tool for generating efficient hardware assertion checkers is required. This thesis has introduced the methods and algorithms behind checker generation for hardware assertions, and has also introduced a variety of enhancements to the checkers for improving the debugging process and usability in assertion-based verification. It is also shown how checkers can be used to facilitate the debugging of post-fabricated silicon, and also as permanent additions for on-line monitoring. Since assertions can be converted into circuit-level checkers, any simulator can easily be made to support assertions as well.

A two-level symbol alphabet was devised and is used at the core of the automaton-based approach. Minimization and nondeterminism were also shown to play an important role in producing resource-efficient checkers. The nondeterminism brought upon by the symbolic alphabet encoding was also a major point taken into account in the development of the automata algorithms. The semantics of all implementations, whether by algorithms or rewrite rules, was particularized for optimal error reporting as required in run-time verification scenarios.

The implementation of sequential-extended regular expressions required the development of particular algorithms for fusion and length-matching intersection, operators that are not typically used in conventional regular expressions. Sugaring rules in sequences helped to reduce the number of operators that needed to be supported in the kernel of the checker generator. Although these sugaring rules were used directly as rewrite rules, the same approach was generally not possible for the sugaring operators in properties. Restrictions are made to the full language of PSL to obtain properties suitable for dynamic verification, with non-branching time. A set of rewrite rules was devised to implement the majority of PSL's large array of property operators. This was an important aspect that allowed all of the PSL operators to be supported in the tool.

In the debugging enhancements portion of the research, multiple additions and modification were proposed to facilitate debugging with checkers. Activity monitors play a key role in observing trivial validity, and completion mode helps to build confidence in the coverage of the test suite. Assertion and cover counters help build more detailed result metrics, and signal dependencies help converge to the true cause of an assertion failure by indicating the relevant signals used in an assertion. A more efficient implementation of the eventually! operator, compared to the rewrite rule, was also developed and was based on automata splitting and the use of logic gates mixed with automata.

Extending the use of checkers beyond pre-fabrication verification was also explored by presenting various on-line and off-line self test scenarios based on the inclusion of assertion checkers. Post-fabricated silicon can benefit from the inclusion of checkers, whereby assertions can be exercised under realistic operating conditions, where timing issues can be truly revealed. The idea of using a checker generator and assertions as a means of performing certain types of circuit design was also put forth, where an example scenario in redundancy control was shown.

Many of the themes explored in this work can be applied to other assertion languages as well, and have already been applied to checker generation of SystemVerilog Assertions [34]. Sequential regular expressions are used in many verification languages, and the automata framework and algorithms devised apply to those cases as well. The rewrite rules developed for properties can also be used in other verification tools to allow a quick implementation of the majority of operators.

In general, adding checkers to a Device Under Verification (DUV) can not be done without impacting the timing of the signals in the original circuit. The effects on specific DUV circuits were not explored in this thesis, and the emphasis was placed on the complexity of checkers that are instrumented within them. When comparing to other tools, smaller assertion checkers lessen the negative impacts on the DUV.

An upper bound on the area penalty of checkers in hardware is obtained by assuming that no sharing of common circuit primitives takes place. Since the checkers only monitor the internal circuit signals, the extra loading can at worst add small delays, which can be kept low by following standard design techniques. For instance, for the signals in the critical path that are monitored by an assertion, small buffers can be inserted to minimize the loading of the circuit under debug.

This thesis has shown how to create resource-efficient circuit-level checkers from assertions, for use in hardware verification, silicon debugging and on-line monitoring. Compared to the only available checker generator, the MBAC checker generator shows important improvements in terms of the resource usage, behavior and capability of the assertion circuits that are produced. The circuit size of checkers is a particularly important parameter when multiple checkers are to be included in valuable silicon area.

As assertions become ubiquitous in design verification, silicon debugging, on-line monitoring, and also as potential new ways of performing design specification, the ability to synthesize assertions into circuits will continue to grow in importance in the field of electronic design automation. With human ambition constantly pushing back the bounds of design complexity, we should not forget that the task of producing quality electronic designs will also become increasingly more challenging.

## 8.2 Future Work

The research on checker generation that was performed in this work can be extended in a variety of directions. The following lists show examples of particular aspects that can be continued in future research, and are grouped into five themes.

### 8.2.1 Optimizations and Improvements

The efficient implementation of eventually! leads to a few thoughts regarding optimizations in the checker generator, while other proposed optimizations are related to automata and counters.

- The fourth assertion in the up-down counter example (Example 2.7) is of the form assert never $b$[*10];, where $b$ is a Boolean expression. The implementation requires ten states, thus ten flip-flops are used. However, because this assertion is non-overlapping, i.e. there are no activations coming from an arbitrarily occurring antecedent, a four bit counter could be used instead. In this example the counter would count the number of consecutive $b$s, and if 10 or more are seen, an error would be signaled. Such optimizations could be implemented, and were mentioned also in a document pertaining to the PROSYD project [106], where optimizing checkers with non-overlapping instances is proposed.

- The more efficient implementation of eventually! required the use of automata splitting, and was shown to be an improvement to the rewrite rule also developed for this operator. However, when the user enters a property using the same syntax as the right side of the rewrite rule, namely {[+]:*seq*}!, the tool could be made to automatically detect this and also switch to the technique using splitting.
- The efficient implementation of eventually! using automata splitting was made possible by the addition of logic gates to the signals in the automata. Further exploration might reveal other instances where breaking up automata and using logic around these could produce smaller checkers.
- Table 7.10 (and many other tables in Chapter 7) revealed that although two different checkers may be proven to be functionally equivalent, the synthesis tools are not always able to reduce both to circuits of the same size. In these examples, the checkers produced as a result of using the rewrite rule were larger. Logic then dictates that if the checker generator was able to produce one version that yielded a smaller circuit than its equivalent other version, then the checker generator must be doing an optimization that the synthesis tool is not doing. This fact could be the starting point for perhaps more optimizations in synthesis tools.
- In general, NFAs are more compact than their equivalent DFAs. Adding a form of DFA to NFA conversion could help produce even smaller checkers; however, converting DFAs to NFAs and minimizing NFAs [117] is not computationally efficient, not to mention a hard problem [115]. A heuristic approach could perhaps allow some form of nondeterminism to be applied even if not complete. DFA to NFA conversion could benefit the automata that undergo weak determinization (in minimization), and also those that undergo strong determinization (in failure matching).

### 8.2.2 Checkers and Debugging

The debug enhancements presented in Section 6.4 can be furhter extended as proposed below. Another example application of checkers is also proposed, for use in embedded logic analyzers.

- The assertion threading enhancement could benefit from more aggressive evaluations in the checker generator where multiple aspects of assertion threading could be automatically controlled. For example, automatically calculating the appropriate number of threads to instantiate could be performed. If sequences are finite, then perfect separation of concurrent activity could be achieved. If infinite sequences are to be threaded, an estimation could be made to trade-off circuit size with visibility. Currently, the number of threads to be used is controlled via a command line parameter to the tool.

- In the same line of thought as the previous point, assertion threading could also be augmented with concurrency detection logic, to show when activity is being overlapped in a thread. For example, if an activation enters a thread that is still processing the previous activation, no guarantees can be made as to the order of the failures, therefore determining the exact cause of a failure is more difficult.
- Currently, top-level sequences can be threaded and/or monitored. Activity monitors could be generalized for arbitrary sub-properties and sub-sequences, so that any valid sub-expression in an assertion could be monitored. For example, in the property always $a$ -> $p$ until $b$, generating an activity signal for property $p$ could allow it to be checked for trivial validity. Generalizing assertion threading in the same manner could also be explored. The difficulty with these types of refinements is that after applying automata operators such as sequence intersection, the argument automata are no longer intact and separable. This makes monitoring arbitrary sub-expressions more challenging.
- The checker generator could be a central component in a debugging tool that would automate the instrumentation of embedded logic analyzers into designs under test. The assertion checkers would play the role of implementing the triggering mechanism. In this way, the user could specify triggering conditions using assertions, and the checkers would be instrumented as part of the embedded trace memory and would serve to control the sampling points.

### 8.2.3 Testing the Checkers

Test generation and testbenches are essential when testing the checkers themselves, in dynamic verification; related developments include:

- It would also be important to develop universally accepted set of benchmark assertions for evaluating the performance of checker generators. The test assertions would be shared between researchers and would be used extensively for benchmarking, analogous to the TPTP library [164] (Thousands of Problems for Theorem Provers), used in the field of automated theorem proving.

- When the checkers themselves need to be tested in dynamic verification, it is often necessary to drive their inputs such that their output trigger reasonably often. This is useful when two different checkers for the same assertion are exercised to determine behavioral differences, such as in Section 7.6.3 where failure-matching sequences were benchmarked. For the simulation based comparison to be the most meaningful, the testbench often performs biased random simulation, whereby the probabilities of signals are adjusted. With inappropriate signal probabilities, an assertion signal could only trigger a handful of times in a million cycles of simulation, thus not allowing any meaningful information to be deduced. The development of an automatic process to determine signal probabilities would allow greater automation and effectiveness in testbenches, where multiple checkers could be tested in sequence. The determination of random weightings for checker inputs is referred to as probabilistic test generation, as opposed to the explicit test generation that aims to enumerate separate test vectors for stimulating the checkers. Explicit test generation can be performed by traversing the automata representing checkers, or by applying sequential ATPG (Automatic Test Pattern Generation) to circuit-level checkers. Including the DUV in the process would produce the test vectors to exercise the entire design with the checkers, but would require much more computation for any reasonably sized design.

### 8.2.4 Assertion Languages

The assertion languages themselves, whether PSL or SVA, also present some challenges for future developments.

- Many designs make use of multiple clock signals, especially in heterogeneous SoC designs. Adding support for the "@" clocking operator would allow the specification of assertions with multiple clock domains [136]. The support of Boolean clocks that are level sensitive, as opposed to edge sensitive, could also be developed.
- As witnessed in Section 2.4 where the overlapping `until` was relaxed to allow a full property as its left argument, one can be led to the question of whether simple subset restrictions on other operators can be lifted as well. This leads to the idea of developing a formal derivation of precisely what is and what is not simulatable, and what are the precise language bounds that can be placed on PSL while still remaining suitable for dynamic verification.

- Many of the techniques developed in this work also apply to checker generation for SystemVerilog Assertions (SVA); this is reported in the book based on this thesis [34]. The automata framework and symbol alphabet strategy is identical for generating SVA checkers, as well as determinization and minimization. The disable iff operator in SVA is very similar to PSL's abort operator, and both forms of suffix implication are identical in both languages. Property conjunction and disjunction are also similar; however, the disjunction operator allows both arguments to be properties. Negation is also defined for full SVA properties, and combined with the more expressive disjunction operator, has required the development of a new strategy whereby negations are pushed down to sequences. Many interesting rules are developed, notably an application of De Morgan's law applied to temporal sub-properties and their automata. SVA sequences have many similarities with PSL sequences, and the suite of automata operators can be readily used to map SVA sequences into PSL sequences. All that must be added is a particular algorithm to handle the SVA cycle delay operator (##$n$ or ##[$n$:$m$]). The FirstMatch() algorithm (Algorithm 6.1) can be used to implement the SVA sequence operator of the same name. Exploring the mapping of SVA into PSL [96] offers some insights into adapting the PSL checker generator kernel for another assertion language, although the translation is not always direct. For example, recursive property declarations are allowed in SVA, and are used to model certain types of temporal operators. In SVA, local variables can be assigned and sampled within assertions, and have no equivalent in PSL. The implementation of local variables in SVA for formal verification [124] could be a starting point for their implementation in hardware checkers for dynamic verification. Although the MBAC checker generator does currently handle SVA statements, the support for local variables and recursive property declarations remain as future challenges [34].

### 8.2.5 Beyond RTL Assertion Languages

Applying the concepts in this thesis beyond RTL (Register Transfer Level) assertion languages also presents opportunities to further enlarge the scope of this research.

- The use of PSL and a checker generator could be explored for use in high-level synthesis, similar to the Production Based Specification work [160] and the high-level synthesis of synchronous languages such as Esterel [66]. Exploiting the high-level expressiveness of PSL and the circuit-synthesis capabilities of the checker generator could lead to new high-level hardware design practices.

- It would also be interesting to see how the developments introduced in this thesis could be applied to create checkers for transaction-level assertions [65].

# Appendix A

# Example for Up-down Counter

In this appendix the source files used in the up-down counter studied in Example 2.7 are shown. The counter to be verified is shown below in the Verilog language. The assertions used to verify the counter are coded in PSL in the *udcounter.psl* file that follows. The third file in this example contains the assertion checkers produced by the checker generator (*udcounter_psl.v*). The checkers are also expressed in Verilog HDL, and allow the assertions to be embedded in the circuit under verification during hardware emulation in the verification stages, or in the fabricated integrated circuit for at-speed silicon debugging.

The command given to the checker generator in this example is as follows:

MBAC udcounter.v udcounter.psl

**udcounter.v**:

```
module udcounter(cnt, load, en_load, en_ud, up_ndown, clk, reset);
  parameter width = 8;
  output reg [width-1:0] cnt;
  input [width-1:0] load;
  input en_load, en_ud, up_ndown, clk, reset;

  always @(posedge clk)
    if (!reset)
      cnt <= 0;
    else if (en_load)
      cnt <= load;
    else if (en_ud)
```

```
      if (up_ndown)
        cnt <= cnt+1;
      else
        cnt <= cnt-1;
endmodule //udcounter
```

**udcounter.psl**:

```
vunit vu1(udcounter){
  default clock = (posedge clk);

  //if no count nor load, value should not change
  property astab = always {~en_ud & ~en_load} |=> stable(cnt);
  assert astab;

  //ensure load works
  property ald = always en_load -> next (cnt == prev(load));
  assert ald;

  //no roll-over
  property nro = always ~en_load -> next (!(cnt == ~prev(cnt) &&
    cnt[width-1]==cnt[0]));
  assert nro;

  //no inactivity
  property ninac = never (~en_load & ~en_ud)[*10];
  assert ninac;
}
```

**udcounter_psl.v**:

```
//Generated by MBAC v1.75
//15-6-2007, 15h 18m 49s
//---------------------------------------------

//RESET_POLARITY_SYMBOL, set to ! (or blank) for active low (high)
```

```
‘define MBACRPS !

//Assertion circuit for vunit: vu1
//vunit is bound to module: udcounter
module udcounter_psl_vu1 (udcounter_psl_vu1_out, reset, clk, en_ud,
  en_load, cnt, load);
  parameter width = 8;
  output [4:1] udcounter_psl_vu1_out;
  input reset, clk, en_ud, en_load;
  input [width - 1:0] cnt;
  input [width - 1:0] load;

  reg [width - 1:0] s1;
  wire [2:0] s3s;
  reg [2:0] s3sq;
  reg [width - 1:0] s4;
  wire [2:0] s5s;
  reg [2:0] s5sq;
  reg [width - 1:0] s6;
  wire [2:0] s7s;
  reg [2:0] s7sq;
  wire [10:0] s8s;
  reg [10:0] s8sq;
  wire s2;
  reg ASR_1, ASR_2, ASR_3, ASR_4;

  assign udcounter_psl_vu1_out={ASR_4, ASR_3, ASR_2, ASR_1};

  //---------------
  //ASR_1 : assert always {~ en_ud & ~ en_load} |=> stable(cnt);
  //---------------
  always @(posedge clk) if (‘MBACRPS reset) s1<=0; else
    s1<=cnt;
  assign s2 = s1 == cnt;
  always @(posedge clk) if (‘MBACRPS reset) s3sq<=3’h4; else
    s3sq<=s3s;
```

```
assign s3s={1'b1,
  ((~ en_ud) & (~ en_load)),
  (s3sq[1] & ~(s2))};
always @(posedge clk) if (`MBACRPS reset) ASR_1<=0; else
  ASR_1 <= (s3s[0]);

//---------------
//ASR_2 : assert always en_load -> next ( cnt == prev(load) );
//---------------
always @(posedge clk) if (`MBACRPS reset) s4<=0; else
  s4<=load;
always @(posedge clk) if (`MBACRPS reset) s5sq<=3'h4; else
  s5sq<=s5s;
assign s5s={1'b1,
  en_load,
  (s5sq[1] & ~((cnt == s4)))};
always @(posedge clk) if (`MBACRPS reset) ASR_2<=0; else
  ASR_2 <= (s5s[0]);

//---------------
//ASR_3 : assert always ~ en_load -> next ( ! ( cnt == ~ prev(cnt)
           && cnt[width - 1] == cnt[0] ) );
//---------------
always @(posedge clk) if (`MBACRPS reset) s6<=0; else
  s6<=cnt;
always @(posedge clk) if (`MBACRPS reset) s7sq<=3'h4; else
  s7sq<=s7s;
assign s7s={1'b1,
  ~(en_load),
  (s7sq[1] & ((cnt == (~ s6)) && (cnt[width - 1] == cnt[0])))};
always @(posedge clk) if (`MBACRPS reset) ASR_3<=0; else
  ASR_3 <= (s7s[0]);

//---------------
//ASR_4 : assert never ( ~ en_load & ~ en_ud )[*10];
//---------------
```

```
  always @(posedge clk) if (‘MBACRPS reset) s8sq<=11’h400; else
    s8sq<=s8s;
  assign s8s={1’b1,
    ((~ en_load) & (~ en_ud)),
    (s8sq[9] & ((~ en_load) & (~ en_ud))),
    (s8sq[8] & ((~ en_load) & (~ en_ud))),
    (s8sq[7] & ((~ en_load) & (~ en_ud))),
    (s8sq[6] & ((~ en_load) & (~ en_ud))),
    (s8sq[5] & ((~ en_load) & (~ en_ud))),
    (s8sq[4] & ((~ en_load) & (~ en_ud))),
    (s8sq[3] & ((~ en_load) & (~ en_ud))),
    (s8sq[2] & ((~ en_load) & (~ en_ud))),
    (s8sq[1] & ((~ en_load) & (~ en_ud)))};
  always @(posedge clk) if (‘MBACRPS reset) ASR_4<=0; else
    ASR_4 <= (s8s[0]);

endmodule //udcounter_psl_vu1
/*Instantiation code:
udcounter_psl_vu1 #(width) i_udcounter_psl_vu1 (udcounter_psl_vu1_out,
reset, clk, en_ud, en_load, cnt, load);
*/
//End of circuit(s) for vunit: vu1
```

# Bibliography

[1] Yael Abarbanel, Ilan Beer, Leonid Glushovsky, Sharon Keidar, and Yaron Wolfsthal. FoCs: Automatic Generation of Simulation Checkers from Formal Specifications. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, pages 538–542, 2000.

[2] Miron Abramovici, Paul Bradley, Kumar Dwarakanath, Peter Levin, Gerard Memmi, and Dave Miller. A Reconfigurable Design-for-Debug Infrastructure for SoCs. In *Proceedings of the 43rd Design Automation Conference (43rd DAC)*, pages 7–12, 2006.

[3] Miron Abramovici, Melvin Breuer, and Arthur Friedman. *Digital Systems Testing & Testable Design.* Computer Science Press, New York, New York, 1990.

[4] Hussain Al-Asaad, Brian Murray, and John P. Hayes. Online BIST for Embedded Systems. *IEEE Design & Test of Computers*, 15(4):17–24, 1998.

[5] Thomas Anantharaman, Edmund Clarke, Michael Foster, and Bud Mishra. Compiling Path Expressions into VLSI Circuits. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 191–204, 1985.

[6] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Yael Zbar. The ForSpec Temporal Logic: A New Temporal Property-Specification Language. In *Proceedings of the 2002 International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2002)*, pages 296–211, 2002.

[7] Ilan Beer, Shoham Ben-David, Cindy Eisner, Yechiel Engel, Raanan Gewirtzman, and Avner Landver. Establishing PCI Compliance Using Formal Verifi-

cation: a Case Study. In *Proceedings of the 14th Annual IEEE International Phoenix Conference on Computers and Communications*, pages 373–377, 1995.

[8] Ilan Beer, Shoham Ben-David, Cindy Eisner, Dana Fisman, Yoav Rodeh, and Anna Gringauze. The Temporal Logic Sugar. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, pages 363–368, 2001.

[9] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Avner Landver. RuleBase: An Industry-Oriented Formal Verification Tool. In *Proceedings of the 33rd Design Automation Conference (33rd DAC)*, pages 655–660, 1996.

[10] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. Efficient Detection of Vacuity in Temporal Model Checking. *Formal Methods in System Design*, 18(2):141–163, 2001.

[11] Ilan Beer, Shoham Ben-David, Daniel Geist, Raanan Gewirtzmann, and Michael Yoeli. Methodology and System for Practical Formal Verification of Reactive Hardware. In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV'94)*, pages 182–193, 1994.

[12] Ilan Beer, Shoham Ben-David, and Avner Landver. On-the-Fly Model Checking of RCTL Formulas. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98)*, pages 184–194, 1998.

[13] Janick Begeron. *Writing Testbenches: Functional Verification of HDL Models*. Springer-Verlag, New York, New York, second edition, 2003.

[14] Janick Begeron. *Verification Methodology Manual for SystemVerilog*. Springer-Verlag, New York, New York, 2005.

[15] Janick Begeron. *Writing Testbenches using SystemVerilog*. Springer-Verlag, New York, New York, 2006.

[16] Shoham Ben-David. Syntactic Sugaring of CTL Formulas as a Productivity Aid to Formal Verification. Technical report, IBM, 1995.

[17] Shoham Ben-David, Dana Fisman, and Sitvanit Ruah. Automata Construction for Regular Expressions in Model Checking. Technical Report H-0229, IBM, 2004.

[18] Shoham Ben-David, Dana Fisman, and Sitvanit Ruah. Embedding Finite Automata within Regular Expressions. In *Proceedings of the 1st International Symposium on Leveraging Applications of Formal Methods (ISOLA'04)*, 2004.

[19] Shoham Ben-David, Dana Fisman, and Sitvanit Ruah. The Safety Simple Subset. In *Proceedings of the 1st International Haifa Verification Conference (HVC'05)*, pages 14–29, 2005.

[20] Victor Berman, Erich Marschner, and Lisa Piper. Pragmatic ABV: Effective Assertion-Based Verification. *Design & Verification Conference (DVCon'05), Tutorial 2*, 2005.

[21] Gérard Berry and Ravi Sethi. From Regular Expressions to Deterministic Automata. *Theoretical Computer Science*, 48(1):117–126, 1986.

[22] Jayaram Bhasker. *A Verilog HDL Primer*. Star Galaxy Publishing, Allentown, Pennsylvania, 1997.

[23] Dominique Borrione. Personal communications. November, 2006.

[24] Dominique Borrione, Miao Liu, Katell Morin-Allory, Pierre Ostier, and Laurent Fesquet. On-Line Assertion-Based Verification with Proven Correct Monitors. In *Proceedings of the 3rd ITI International Conference on Information & Communications Technology (ICICT 2005)*, pages 123–143, 2005.

[25] Dominique Borrione, Miao Liu, Pierre Ostier, and Laurent Fesquet. PSL-Based Online Monitoring of Digital Systems. In A. Vachoux, editor, *Applications of Specification and Design Languages for SoCs*, chapter 1. Springer, 2005.

[26] Marc Boulé, Jean-Samuel Chenard, and Zeljko Zilic. Adding Debug Enhancements to Assertion Checkers for Hardware Emulation and Silicon Debug. In *Proceedings of the 24th IEEE International Conference on Computer Design (ICCD'06)*, pages 294–299, 2006.

[27] Marc Boulé, Jean-Samuel Chenard, and Zeljko Zilic. Checkers in Verification, Silicon Debug and In-Field Diagnosis. In *Proceedings of the 8th International Symposium on Quality Electronic Design (ISQED'07)*, pages 613–618, 2007.

[28] Marc Boulé, Jean-Samuel Chenard, and Zeljko Zilic. Debug Enhancements in Assertion-Checker Generation. *IET Computers and Digital Techniques – Special Issue on Silicon Debug and Diagnosis*, 1(6):669–677, 2007.

[29] Marc Boulé and Zeljko Zilic. Incorporating Efficient Assertion Checkers into Hardware Emulation. In *Proceedings of the 23rd IEEE International Conference on Computer Design (ICCD'05)*, pages 221–228, 2005.

[30] Marc Boulé and Zeljko Zilic. Efficient Automata-Based Assertion-Checker Synthesis of PSL Properties. In *Proceedings of the 2006 IEEE International High Level Design Validation and Test Workshop (HLDVT'06)*, pages 69–76, 2006.

[31] Marc Boulé and Zeljko Zilic. An Automata Unit, a Tool for Designing Checker Circuitry and a Method of Manufacturing Hardware Circuitry Incorporating Checker Circuitry. *Filed for US patent, US 11/864,030*, September 2007.

[32] Marc Boulé and Zeljko Zilic. Efficient Automata-Based Assertion-Checker Synthesis of SEREs for Hardware Emulation. In *Proceedings of the 12th Asia and South Pacific Design Automation Conference (ASP-DAC2007)*, pages 324–329, 2007.

[33] Marc Boulé and Zeljko Zilic. Automata-Based Assertion-Checker Synthesis of PSL Properties. *ACM Transactions on Design Automation of Electronic Systems (ACM-TODAES)*, 13(1), 2008. Article 4.

[34] Marc Boulé and Zeljko Zilic. *Generating Hardware Assertion Checkers – For Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring*. Springer, Dordrecht, Netherlands, 2008. To be published.

[35] Stephan Bourduas, Jean-Samuel Chenard, and Zeljko Zilic. A RTL-Level Analysis of a Hierarchical Ring Interconnect for Network-on-Chip Multi-Processors. In *Proceedings of the International System-on-a-Chip Design Conference (ISOCC'06)*, 2006.

[36] Anne Bruggemann-Klein. Regular Expressions into Finite Automata. *Theoretical Computer Science*, 120(2):197–213, 1993.

[37] Janusz Brzozowski. Canonical Regular Expressions and Minimal State Graphs for Definite Events. *Mathematical Theory of Automata*, 12 of MRI Symposia Series, Polytechnic Press, Polytechnic Institute of Brooklyn, N.Y.:529–561, 1962.

[38] Janusz Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[39] Adam Buchsbaum, Raffaele Giancarlo, and Jeffery Westbrook. On the Determinization of Weighted Finite Automata. *Lecture Notes in Computer Science*, 1443:482–493, 1998.

[40] Doron Bustan, Dana Fisman, and John Havlicek. Automata Construction for PSL. Technical Report MCS05-04, The Weizmann Institute of Science, 2005.

[41] Cadence Design Systems. ATI Technologies Selects Cadence Palladium II for Verification of Advanced DTV Chips. `www.cadence.com/company/newsroom/press_releases/pr.aspx?xml=021505_ati`, 2005.

[42] Cadence Design Systems. Incisive Palladium Family with Incisive XE Software (Datasheet). `www.cadence.com/datasheets/IncisivePalladiumII_ds.pdf`, 2005.

[43] Cadence Design Systems. The Cadence SMV Model Checker. `www.cadence.com/webforms/cbl_software/index.aspx`, 2007.

[44] Chia-Hsiang Chang and Robert Paige. From Regular Expressions to DFA's Using Compressed NFA's. *Theoretical Computer Science*, 178(1-2):1–36, 1997.

[45] Kai-Hui Chang, Wei-Ting Tu, Yi-Jong Yeh, and Sy-Yen Kuo. A Simulation-Based Temporal Assertion Checker for PSL. In *IEEE International Midwest Symposium on Circuits and Systems (MWSCAS'03)*, 2003.

[46] Pankaj Chauhan, Edmund M. Clarke, Yuan Lu, and Dong Wang. Verifying IP-Core Based System-On-Chip Designs. *IEEE ASIC*, September 1999.

[47] Cyrille Chavet. Modelisation and Validation of a Chip Embeded Architecture for Secure Applications. Master's thesis, TIMA Laboratory - VDS Group, Grenoble France, 2003.

[48] Jean-Samuel Chenard, Stephan Borduas, Nathaniel Azuelos, Marc Boulé, and Zeljko Zilic. Hardware Assertion Checkers in On-Line Detection of Faults in a Hierarchical-Ring Network-On-Chip. In *Poster Presentation at the Design Automation and Test in Europe (DATE 2007) Workshop on Diagnostic Services in Networks-on-Chips*, 2007.

[49] Alessandro Cimatti, Marco Roveri, Simone Semprini, and Stefano Tonetta. From PSL to NBA: a Modular Symbolic Encoding. In *Proceedings of Formal Methods in Computer Aided Design (FMCAD'06)*, pages 125–133, 2006.

[50] Koen Claessen and Johan Martensson. An Operational Semantics for Weak PSL. In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD'04)*, pages 337–351, 2004.

[51] Edmund Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David Long, Kenneth McMillan, and Linda Ness. Verification of the Futurebus+ Cache Coherence Protocol. In *Proceedings of the 11th International Symposium on Computer Hardware Description Languages and their Applications*, pages 5–20, 1993.

[52] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, Cambridge Massachusetts, 2000.

[53] Ben Cohen, Srinivasan Venkataramanan, and Ajeetha Kumari. *Using PSL/ Sugar for Formal and Dynamic Verification*. VhdlCohen Publishing, Los Angeles, California, 2004.

[54] John Cooley. DVCon'04 Trip Report - A Census of 137 Engineers on Design Verification Tool Use. `www.deepchip.com/posts/dvcon04.html`, 2004.

[55] John Cooley. The 2007 DeepChip Verification Census - A Census of 818 Engineers on Design Verification Tool Use. `www.deepchip.com/posts/dvcon07.html`, 2007.

[56] Thomas Cormen, Chaarles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. McGraw-Hill Book Company, 1999.

[57] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007.

[58] Anat Dahan, Daniel Geist, Leonid Gluhovsky, Dmitry Pidan, Gil Shapir, Yaron Wolfsthal, Lyes Benalycherif, Romain Kamdem, and Younes Lahbib. Combining System Level Modeling with Assertion Based Verification. In *Proceedings of the 6th International Symposium on Quality of Electronic Design (ISQED 2005)*, pages 310–315, 2005.

[59] Marco Daniele, Fausto Giunchiglia, and Moshe Vardi. Improved Automata Generation for Linear Temporal Logic. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, pages 249–260, 1999.

[60] Sayantan Das, Rizi Mohanty, Pallab Dasgupta, and Partha Chakrabarti. Synthesis of System Verilog Assertions. In *Proceedings of the 2006 Conference on Design Automation and Test in Europe (DATE'06)*, pages 70–75, 2006.

[61] Kausik Datta and P.P. Das. Assertion Based Verification Using HDVL. In *Proceedings of the 17th International Conference on VLSI Design (VLSI Design 2006)*, pages 319–325, 2004.

[62] Nachum Dershowitz and Zohar Manna. Proving Termination with Multiset Orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[63] Christophe Devaucelle. Formal Verification of a Framer OC-768 (40 Gbits/s). Technical report, IBM Micro-electronics, Essones Labs, Corbeil-Essonnes, France, 2001.

[64] Matthew Dwyer, George Avrunin, and James Corbett. Property Specification Patterns for Finite-State Verification. In *Proceedings of the 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, pages 7–15, 1998.

[65] Wolfgang Ecker, Volkan Esen, Thomas Steininger, Michael Velten, and Michael Hull. Specification Language for Transaction Level Assertions. In *Proceedings of the 2006 IEEE International High Level Design Validation and Test Workshop (HLDVT'06)*, pages 77–84, 2006.

[66] Stephen Edwards. High-level Synthesis from the Synchronous Language Esterel. In *Proceedings of the International Workshop on Logic and Synthesis (IWLS)*, 2002.

[67] Cindy Eisner and Dana Fisman. *A Practical Introduction to PSL*. Springer-Verlag, New York, New York, 2006.

[68] Cindy Eisner, Dana Fisman, and John Havlicek. A Topological Characterization of Weakness. In *Proceedings of the 24th Annual ACM SIGACT-SIGOPS Symposium on Principles Of Distributed Computing (PODC'05)*, pages 1–8, 2005.

[69] Cindy Eisner, Dana Fisman, John Havlicek, Yoad Lustig, Anthony McIsaac, and David Van Campenhout. Reasoning with Temporal Logic on Truncated Paths. In *Proceedings of the 15th Computer-Aided Verification Conference (CAV'03)*, pages 27–39, 2003.

[70] Burak Emir. Compiling Regular Patterns to Sequential Machines. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 1385–1389, 2005.

[71] EVE Corporation. SystemVerilog-Assertion-Based Verification with ZeBu Hardware-Assisted Platform. Synopsys 15th EDA Interoperability Developers' Forum, April 7th 2005. `www.synopsys.com/news/events/devforums/2005/apr/presentations/16_SVA_with_EVE_ZeBu.pdf`.

[72] Dana Fisman. The Subset of Linear Violation. Technical Report MCS05-07, Weizmann Institute of Science, 2005.

[73] Robert Floyd and Jeffrey Ullman. The Compilation of Regular Expressions into Integrated Circuits. *Journal of the ACM (JACM)*, 29(3):603–622, 1982.

[74] Harry Foster, Adam Krolnik, and David Lacey. *Assertion-Based Design*. Kluwer Academic Publishers, Norwell, Massachusetts, second edition, 2004.

[75] Harry Foster, Kenneth Larsen, and Mike Turpin. Introducing The New Accellera Open Verification Library Standard. *Proceedings of the 2006 Design and Verification Conference (DVCon 2006)*, 2006.

[76] Harry Foster, Erich Marschner, and Yaron Wolfsthal. IEEE 1850 PSL: The Next Generation. *Proceedings of the 2005 Design and Verification Conference (DVCon 2005)*, Session 4, Paper 2, 2005.

[77] Ambar Gadkari and S. Ramesh. Automated Synthesis of Assertion Monitors using Visual Specifications. In *Proceedings of the 2005 Conference on Design Automation and Test in Europe (DATE'05)*, pages 390–395, 2005.

[78] Subbu Ganesan. Supplementing VCS-based Simulations with High-Performance Hardware-Assisted Verification. *The Synopsys Verification Avenue Technical Bulletin*, 4(4), 2004.

[79] Eric Gascard. From Sequential Extended Regular Expressions to Deterministic Finite Automata. In *Proceedings of the 3rd ITI International Conference on Information and Communications Technology (ICICT 2005)*, pages 145–157, 2005.

[80] Amjad Gawanmeh, Ali Habibi, and Sofiene Tahar. Embedding and Verification of PSL using AsmL. In *Proceedings of the 12th International Workshop on Abstract State Machines (ASM 2005)*, pages 201–216, 2005.

[81] Daniel Geist, Avner Landver, and Bruce Singer. Formal Verification of a Processor's Bus Interface. Technical report, IBM, August 8th 1996.

[82] Stefan-Valentin Gheorghita. The Art of Translating Sugar to an Automata Language. Master's thesis, Politehnica University of Bucharest, Romania, 2003.

[83] Stefan-Valentin Gheorghita and Radu Grigore. Constructing Checkers from PSL Properties. In *Proceedings of the 15th International Conference on Control Systems and Computer Science (CSCS15)*, volume 2, pages 757–762, 2005.

[84] Amit Goel and William Lee. Formal Verification of an IBM CoreConnect Processor Local Bus Arbiter Core. In *Proceedings of the 37th Design Automation Conference (37th DAC)*, pages 196–200, 2000.

[85] Mike Gordon. Using HOL to Study Sugar 2.0 Semantics. In *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOL 2002)*, pages 87–100, 2002.

[86] Mike Gordon. Validating the PSL/Sugar Semantics Using Automated Reasoning. *Formal Aspects of Computing*, 15(4):406–421, 2003.

[87] Mike Gordon. PSL Semantics in Higher Order Logic. In *Proceedings of the 5th International Workshop on Designing Correct Circuits (DCC'06)*, 2004.

[88] Mike Gordon, Joe Hurd, and Konrad Slind. Executing the Formal Semantics of the Accellera Property Specification Language by Mechanised Theorem Proving. In D. Geist and E. Tronci, editors, *Correct Hardware Design and Verification Methods, LNCS 2860*, pages 200–215. Springer-Verlag, Oct. 2003.

[89] Daniel Grobe and Rolf Drechsler. Checkers for SystemC Designs. In *2nd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2004)*, pages 171–178, 2004.

[90] Ali Habibi, Amjad Gawanmeh, and Sofiene Tahar. Assertion Based Verification of PSL for SystemC Designs. In *Proceedings of the 2004 International Symposium on System-on-Chip*, pages 177–180, 2004.

[91] Ali Habibi and Sofiene Tahar. Design for Verification of SystemC Transaction Level Models. In *Proceedings of the 2005 Conference on Design Automation and Test in Europe (DATE'05)*, pages 560–565, 2005.

[92] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous Observers and the Verification of Reactive Systems. In *Proceedings of the 3rd International Conference on Algebraic Methodology and Software Technology (AMAST'93)*, pages 83–96, 1993.

[93] Hesham Hallal, Alex Petrenko, Andreas Ulrich, and Sergiy Boroday. Using SDL Tools to Test Properties of Distributed Systems. In *Proceeddings of the Formal Approches to Testing of Software (FATES'01), Workshop of the International Conference on Concurrency Theory (CONCUR'01)*, pages 125–140, 2001.

[94] Jounaidi Ben Hassen and Sofiene Tahar. Formal Verification of an SoC Platform Protocol Converter. In *Proceedings of the 2004 IEEE International Symposium on Circuits and Systems (ISCAS 2004)*, volume 5, pages 313–316, 2004.

[95] John Havlicek, Dana Fisman, and Cindy Eisner. Basic Results on the Semantics of Accellera PSL 1.1 Foundation Language. Technical Report 2004.02, Accellera, 2004.

[96] John Havlicek, Dana Fisman, Cindy Eisner, and Erich Marschner. Mapping SVA to PSL. `www.eda.org/vfv/docs/mapping.pdf`, 2003.

[97] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design, San Francisco, California, third edition, 2003.

[98] Dirk Hoffmann, Jurgen Ruf, Thomas Kropf, and Wolfgang Rosenstiel. Simulation Meets Verification - Checking Temporal Properties in SystemC. In *Proceedings of the 26th EUROMICRO Conference (EUROMICRO'00)*, volume 1, pages 1435–1438, 2000.

[99] Kay Hofmann, Philipp Bucher, Laurent Falquet, and Amos Bairoch. The PROSITE Database, its Status in 1999. *Nucleic Acids Research*, 27(1):215–219, 1999.

[100] John Hopcroft. An $n \log n$ Algorithm for Minimizing the States in a Finite Automaton. In Z. Kohavi and A. Paz, editors, *The Theory of Machines and Computations*, pages 189–196. Academic Press, New York, 1971.

[101] John Hopcroft, Rajeev Motwani, and Jeffrey Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, second edition, 2000.

[102] Yu-Chin Hsu, Bassam Tabbara, Yimg-An Chen, and Furshing Tsai. Advanced Techniques for RTL Debugging. In *Proceedings of the 40th Design Automation Conference (40th DAC)*, pages 362–367, 2003.

[103] Alan Hu, Jeremy Casas, and Jin Yang. Efficient Generation of Monitor Circuits for GSTE Assertion Graphs. In *Proceedings of the 2003 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'03)*, pages 154–159, 2003.

[104] B. L. Hutchings, R. Franklin, and D. Carver. Assisting Network Intrusion Detection with Reconfigurable Hardware. In *Proceedings of the 10th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2002)*, pages 111–120, 2002.

[105] IBM. PSL/Sugar Parser. `www.haifa.il.ibm.com/projects/verification/sugar/parser.html`, 2004.

[106] IBM. Optimized Algorithms for Dynamic Verification. *Property Based System Design (PROSYD) - Deliverable 3.2/5*, `www.prosyd.org/twiki/view/Public/DeliverablePageWP3`, 2005.

[107] IBM. Property-by-Example Guide: a Handbook of PSL/Sugar Examples. *Property-Based System Design (PROSYD), Deliverable 1.1/3*, `www.prosyd.org/twiki/view/Public/DeliverablePageWP1`, 2005.

[108] IBM AlphaWorks. FoCs Property Checkers Generator, version 2.04. `www.alphaworks.ibm.com/tech/FoCs`, 2007.

[109] IEEE Std. 1647-2006. *IEEE Standard for the Functional Verification Language 'e'*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, USA, 2006.

[110] IEEE Std. 1800-2005. *IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, USA, 2005.

[111] IEEE Std. 1850-2005. *IEEE Standard for Property Specification Language (PSL)*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, USA, 2005.

[112] IEEE Std. 1850-200x Working Group. Simple Subset Issue #99, Group E.1. *Issues To Be Addressed in IEEE 1850-200x PSL*, 2006.

[113] IEEE Std. 1850-200x Working Group. Unaddressed Issue #146. *Issues To Be Addressed in IEEE 1850-200x PSL*, 2006.

[114] ISO 8402:1995. *Quality Management and Quality Assurance.* International Organization for Standardization, Geneva, Switzerland, 1995.

[115] Tao Jiang and B. Ravikumar. Minimal NFA Problems are Hard. *SIAM Journal on Computing*, 22(6):1117–1141, 1993.

[116] Cliff Jones. The Early Search for Tractable Ways of Reasoning about Programs. *IEEE Annals of the History of Computing*, 25(2):26–49, 2003.

[117] Tsunehiko Kameda and P. Weiner. On the State Minimization of Nondeterministic Finite Automata. *IEEE Transactions on Computers*, C-19(7):617–627, 1970.

[118] Christian Josef Kargl. A Sugar Translator. Master's thesis, Institut für Softwaretechnologie, Technische Univesität Graz, Graz, Austria, 2003.

[119] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.

[120] Thomas Kropf. *Introduction to Formal Hardware Verification.* Springer-Verlag, 1999.

[121] Thomas Kuhn. *The Structure of Scientific Revolutions.* University Of Chicago Press, third edition, 1996.

[122] Orna Kupferman and Moshe Vardi. Weak Alternating Automata Are Not That Weak. In *Proceedings of the 5th Israel Symposium on Theory of Computing and Systems (ISTCS'97)*, pages 147–158, 1997.

[123] Orna Kupferman and Moshe Vardi. Vacuity Detection in Temporal Model Checking. In *Conference on Correct Hardware Design and Verification Methods*, pages 82–96, 1999.

[124] Jiang Long and Andrew Seawright. Synthesizing SVA Local Variables for Formal Verification. In *Proceedings of the 44th Design Automation Conference (44th DAC)*, pages 75–80, 2007.

[125] Erich Marschner, Bernard Deadman, and Grant Martin. IP Reuse Hardening via Embedded Sugar Assertions. *IP Based SoC Design*, 2002.

[126] Amir Masoud Gharehbaghi, Benyamin Hamdin Yaran, Shaahin Hessabi, and Maziar Goudarzi. An Assertion-Based Verification Methodology for System-Level Design. *Computers and Electrical Engineering*, 33(4):269–284, 2007.

[127] Kenneth McMillan. *Symbolic Model Checking*. Springer, 1993.

[128] Robert McNaughton and H. Yamada. Regular Expressions and State Graphs for Automata. *IEEE Transactions on Electronic Computers*, EC-9(1):39–47, 1960.

[129] Mentor Graphics. ModelSim SE 6.1f. `www.mentor.com/products/fv/digital_verification/modelsim_se/index.cfm`, 2006.

[130] Mentor Graphics. Veloce MHz-Class Accelerator/Emulator. `www.mentor.com/products/fv/emulation/veloce`, 2007.

[131] Mehryar Mohri. Finite-State Transducers in Language and Speech Processing. *Computational Linguistics*, 23(2):269–311, 1997.

[132] Katell Morin-Allory. Personal communications. August to November, 2007.

[133] Katell Morin-Allory and Dominique Borrione. A Proof of Correctness for the Construction of Property Monitors. In *Proceedings of the 2005 IEEE International High Level Design Validation and Test Workshop (HLDVT'05)*, pages 237–244, 2005.

[134] Katell Morin-Allory and Dominique Borrione. On-Line Monitoring of Properties Built on Regular Expression Sequences. *Forum on Specification & Design Languages (FDL'06)*, 2006.

[135] Katell Morin-Allory and Dominique Borrione. Proven Correct Monitors from PSL Specifications. In *Proceedings of the 2006 Conference on Design Automation and Test in Europe (DATE'06)*, pages 1246–1251, 2006.

[136] Katell Morin-Allory, Laurent Fesquet, and Dominique Borrione. Asynchronous Assertion Monitors for Multi-Clock Domain System Verification. In *Proceedings of the Seventeenth IEEE International Workshop on Rapid System Prototyping (RSP'06)*, pages 98–102, 2006.

[137] Anindyasundar Nandi, Bhaskar Pal, Nawang Chhetan, Pallab Dasgupta, and Partha Chakrabarti. H-DBUG: A High-level Debugging Framework for Protocol Verification using Assertions. In *Proceedings of the 2005 Annual IEEE INDICON Conference*, pages 115–118, 2005.

[138] Gonzalo Navarro and Mathieu Raffinot. Fast and Simple Character Classes and Bounded Gaps Pattern Matching, With Application to Protein Searching. In *Proceedings of the Fifth International Annual Conference on Computational Molecular Biology*, pages 231–240, 2001.

[139] Kelvin Ng, Alan Hu, and Jin Yang. Generating Monitor Circuits for Simulation-Friendly GSTE Assertion Graphs. In *Proceedings of the 22rd IEEE International Conference on Computer Design (ICCD'04)*, pages 288–492, 2004.

[140] Yann Oddos, Katell Morin-Allory, and Dominique Borrione. Prototyping Generators for On-Line Test Vector Generation Based on PSL Properties. In *Proceedings of the 10th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems (DDECS'07)*, pages 383–388, 2007.

[141] Etienne Ogoubi and Eduard Cerny. Synthesis of Checker EFSMs from Timing Diagram Specifications. In *Proceedings of the 1999 IEEE International Symposium on Circuits and Systems, (ISCAS'99)*, pages 13–18, 1999.

[142] Marcio Oliveira. High-Level Specification and Automatic Generation of IP Interface Monitors. Master's thesis, University of British Columbia, 2003.

[143] Marcio Oliveira and Alan Hu. High-Level Specification and Automatic Generation of IP Interface Monitors. In *Proceedings of the 39th Design Automation Conference (39th DAC)*, pages 129–134, 2002.

[144] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS System Guide, version 2.4. *SRI International, Computer Science Laboratory, Menlo Park, CA*, 2001.

[145] Avi Parash. Formal Verification of an MPEG Decoder Chip. *Integrated System Design*, 12(134):44–55, 2000.

[146] Priyadarsan Patra. On the Cusp of a Validation Wall. *IEEE Design & Test*, 24(2):193–196, 2007.

[147] Michael Pellauer, Mieszko Lis, Don Baltus, and Rishiyur Nikhil. Synthesis of Synchronous Assertions with Guarded Atomic Actions. *3rd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2005)*, pages 15–24, 2005.

[148] Douglas Perry and Harry Foster. *Applied Formal Verification*. McGraw-Hill, New York, New York, 2005.

[149] Kevin Peterson and Yvon Savaria. Assertion-Based On-Line Verification and Debug Environment for Complex Hardware Systems. In *Proceedings of the 2004 International Symposium on Circuits and Systems (ISCAS'04)*, volume 2, pages 685–688, 2004.

[150] Marco Platzner. Reconfigurable Computer Architectures - Rekonfigurierbare Rechnerarchitekturen. `citeseer.ist.psu.edu/490784.html`.

[151] Marco Platzner. Reconfigurable accelerators for combinatorial problems. *IEEE Computer*, 33(4):58–60, 2000. `citeseer.ist.psu.edu/483247.html`.

[152] Pascal Raymond. Recognizing Regular Expressions by Means of Dataflow Networks. In *Proceedings of the 23rd International Colloquium on Automata, Languages and Programming (ICALP'96)*, pages 336–347, 1996.

[153] Sitvanit Ruah, Dana Fisman, and Shoham Ben-David. Automata Construction for On-The-Fly Model Checking PSL Safety Simple Subset. Technical Report H-0234, IBM, 2005.

[154] Jurgen Ruf, Dirk Hoffmann, Thomas Kropf, and Wolfgang Rosenstiel. Checking Temporal Properties Under Simulation of Executable System Descriptions. In *Proceedings of the 2000 IEEE International High Level Design Validation and Test Workshop (HLDVT'00)*, pages 161–166, 2000.

[155] Jurgen Ruf, Dirk Hoffmann, Thomas Kropf, and Wolfgang Rosenstiel. Simulation Guided Property Checking Based on Multi-Valued AR-Automata. In *Proceedings of the 2001 Conference on Design Automation and Test in Europe (DATE'01)*, pages 742–748, 2001.

[156] Jurgen Ruf, Thomas Kropf, and Jochen Klose. A Visual Approach to Validating System Level Designs. In *Proceedings of the 15th International Symposium on System Synthesis (ISSS '02)*, pages 186–191, 2002.

[157] David Russinoff. Formal Verification of Floating-Point RTL at AMD Using the ACL2 Theorem Prover. In *IMACS World Congress Scientific Computation, Applied Mathematics and Simulation*, Paris, France, July 2005.

[158] Andrew Seawright and Forrest Brewer. Synthesis from Production-Based Specification. *Proceedings of the 29th Design Automation Conference (29th DAC)*, pages 194–199, 1992.

[159] Andrew Seawright and Forrest Brewer. High–Level Symbolic Construction Techniques for High Performance Sequential Synthesis. In *Proceedings of the 30th Design Automation Conference (30th DAC)*, pages 424–428, 1993.

[160] Andrew Seawright and Forrest Brewer. Clairvoyant: A Synthesis System for Production-Based Specification. *IEEE Transactions on VLSI Systems*, 2(2):172–185, 1994.

[161] Hiroaki Sengoku. Minimization of Nondeterministic Finite Automata. Master's thesis, Kyoto University, 1992.

[162] A. Udaya Shankar. An Introduction to Assertional Reasoning for Concurrent Systems. *ACM Computing Surveys*, 25(3), 1993.

[163] Reetinder Sidhu and Viktor Prasanna. Fast Regular Expression Matching using FPGAs. In *Proceedings of the 9th IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'01)*, pages 227–238, 2001.

[164] Geoff Sutcliffe and Christian Suttner. The TPTP Problem Library – CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.

[165] Peter Sutton. Partial Character Decoding for Improved Regular Expression Matching in FPGAs. In *Proceedings of the IEEE International Conference on Field-Programmable Technology 2007 (ICFPT'07)*, pages 25–32, 2004.

[166] Synopsys Inc. OpenVera Assertions. `www.synopsys.com/products/simulation/ova_wp.pdf`, 2003.

[167] Bassam Tabbara, Yu-Chin Hsu, George Bakewell, and Scott Sandler. Assertion-Based Hardware Debugging. *Proceedings of the 2003 Design and Verification Conference (DVCon 2003)*, Session 1, Paper 2, 2003.

[168] Hellis Tamm. *On Minimality and Size Reduction of One-Tape and Multitape Finite Automata*. PhD thesis, University of Helsinki, Finland, 2004.

[169] TU-Graz. Automata Construction Algorithms Optimized for PSL. *Property-Based System Design (PROSYD), Deliverable 3.2/4*, `www.prosyd.org/twiki/view/Public/DeliverablePageWP3`, 2005.

[170] TU-Graz. Manual for VIS Tool Port. *Property-Based System Design, Deliverable 3.3/1*, `www.prosyd.org/twiki/view/Public/DeliverablePageWP3`, 2005.

[171] Thomas Tuerk, Klaus Schneider, and Mike Gordon. Model Checking PSL Using HOL and SMV. In *Proceedings of the Second Annual Haifa Verification Conference (HVC'06)*, pages 1–15, 2006.

[172] Gertjan Van Noord. Treatment of $\epsilon$-Moves in Subset Construction. *Computational Linguistics*, 26(1):61–76, 2000.

[173] Moshe Vardi. An automata-theoretic approach to linear temporal logic. In *Banff Higher Order Workshop*, pages 238–266, 1995.

[174] Moshe Vardi. Alternating Automata: Unifying Truth and Validity Checking for Temporal Logics. In *Proceedings of the 14th International Conference on Automated Deduction (CADE-14)*, volume 1249, pages 191–206, 1997.

[175] Moshe Vardi and Pierre Wolper. Reasoning About Infinite Computations. *Information and Computation*, 115(1):1–37, 1994.

[176] Fang Wang, Ali Habibi, and Sofiene Tahar. Translating LTL Specification to MDG-HDL. In *Proceedings of the 2003 IEEE Canadian Conference on Electrical & Computer Engineering (CCECE 2003)*, pages 1369–1373, 2003.

[177] Bruce Watson. A Taxonomy of Finite Automata Minimization Algorithms. Technical Report – Computing Science Note 93/44, Eindhoven University of Technology, 1993.

[178] Bruce Wile, John Goss, and Wolfgang Roesner. *Comprehensive Functional Verification: The Complete Industry Cycle*. Morgan Kaufmann, 2005.

[179] Guangming Xing. A Simple Way to Construct NFA with Fewer States and Transitions. In *Proceedings of the 42nd Annual ACM Southeast Regional Conference*, pages 214–218, 2004.

[180] Chia-Chih Yen and Jing-Yang Jou. An Optimum Algorithm for Compacting Error Traces for Efficient Functional Debugging. In *Proceedings of the 2005 IEEE*

*International High Level Design Validation and Test Workshop (HLDVT'05)*, pages 177–183, 2005.

[181] Aleksandr Zaks and Amir Pnueli. PSL Model Checking and Run-time Verification via Testers. In *Proceedings of the 14th International Symposium on Formal Methods (FM'06)*, pages 573–586, 2006.

[182] Avi Ziv. Cross-Product Functional Coverage Measurement with Temporal Properties-based Assertions. *Proceedings of the 2003 Conference on Design Automation and Test in Europe (DATE'03)*, pages 834–839, 2003.