NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

DISSERTATION

DEFINING AND ENFORCING HARDWARE SECURITY REQUIREMENTS

by

Michael B. Bilzor

December 2011

Dissertation Supervisor: Ted Huffmire

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
16-DEC-2011	Dissertation	APR 2010–DEC 2011

4. TITLE AND SUBTITLE

Defining and Enforcing Hardware Security Requirements

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Bilzor, Michael B.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

U.S. Naval Postgraduate School, 1 University Circle, Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

This research was funded in part by National Science Foundation Grant CNS-0910734.

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. I.R.B. protocol number N.A.

14. ABSTRACT

Security in computing systems to date has focused mostly on software. In this research, we explore the application and enforceability of well-defined security requirements in hardware designs. The principal threats to hardware systems demonstrated in the academic literature to date involve some type of subversion, often called a Hardware Trojan or malicious inclusion. Detecting these has proved very difficult. We demonstrate a method whereby the dynamic enforcement of a processor's security requirements can be used to detect the presence of some of these malicious inclusions. Although there are theoretical limits on which security properties can be dynamically enforced using the techniques we describe, our research does provide a novel method for expressing and enforcing security requirements at runtime in hardware designs. While the method does not guarantee the detection of all possible malicious inclusions in a given processor, it addresses a large class of inclusions—those detectable as violations of behavioral restrictions in the architectural specification—which provides significant progress against the general case, given a suitably complete set of checkers.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(include area code)*
Unclassified	Unclassified	Unclassified	UU	165	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

DEFINING AND ENFORCING HARDWARE SECURITY REQUIREMENTS

Michael B. Bilzor
Commander, United States Navy
B.S., U.S. Naval Academy, 1992
M.S., Johns Hopkins University, 1993

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

from the

UNITED STATES NAVAL POSTGRADUATE SCHOOL
December 2011

Author:

Michael B. Bilzor

Approved By:

Ted Huffmire
Professor of Computer Science
Dissertation Supervisor

Cynthia Irvine
Professor of Computer Science

Tim Levin
Professor of Computer Science

Zachary Peterson
Professor of Computer Science

James Luscombe
Professor of Physics

Approved By:

Peter Denning, Chair, Department of Computer Science

Approved By:

Douglas Moses, Vice Provost for Academic Affairs

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Security in computing systems to date has focused mostly on software. In this research, we explore the application and enforceability of well-defined security requirements in hardware designs. The principal threats to hardware systems demonstrated in the academic literature to date involve some type of subversion, often called a Hardware Trojan or malicious inclusion. Detecting these has proved very difficult. We demonstrate a method whereby the dynamic enforcement of a processor's security requirements can be used to detect the presence of some of these malicious inclusions.

Although there are theoretical limits on which security properties can be dynamically enforced using the techniques we describe, our research does provide a novel method for expressing and enforcing security requirements at runtime in hardware designs. While the method does not guarantee the detection of all possible malicious inclusions in a given processor, it addresses a large class of inclusions—those detectable as violations of behavioral restrictions in the architectural specification—which provides significant progress against the general case, given a suitably complete set of checkers.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

3D-IC	Three Dimensional Integrated Circuit
3PIP	Third-Party Intellectual Property
ABV	Assertion-Based Verification
AMD	Advanced Micro Devices
ASIC	Application-Specific Integrated Circuit
BIOS	Basic Input-Output System
CAD	Computer-Aided Design
CPU	Central Processing Unit
CTL	Computation Tree Logic
DARPA	The Defense Advanced Projects Research Agency
DFF	D-type Flip-Flop
DNF	Disjunctive Normal Form
DoD	The Department of Defense
ELF	Executable and Linkable Format
FIB	Focused Ion Beam
FL	Foundation Language
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Design Language
IBM	International Business Machines Corporation
IC	Integrated Circuit
IEEE	The Institute of Electrical and Electronics Engineers
LTL	Linear Temporal Logic
MI	Malicious Inclusion
NSA	The National Security Agency

OBE	Optional Branching Extensions
PCB	Printed Circuit Board
PLA	Propositional Logic Automaton
PSL	The Property Specification Language
RTL	Register-Transfer Level
SAT	Satisfiability
SERE	Sequence Extended Regular Expression
SMIC	Semiconductor Manufacturing International Corporation
SPARC	Scalable Processor Architecture
SPI	Serial Peripheral Interface
SVA	SystemVerilog Assertions
TIC	Trusted Integrated Circuits
TSV	Through-Silicon Via
UCI	Unused Circuit Identification

GLOSSARY OF TERMS

architectural specification	a processor design document that details the processor's instruction set and functional components, and how they should operate when implemented.
assertion	an evaluatable description of a behavior that is expected to remain true over time.
assertion-based verification	a hardware verification technique involving the application of assertions, such as SystemVerilog Assertions (SVA) or Property Specification Language (PSL) assertions.
covert channel	a mechanism through which information may be transmitted from one entity to another, outside of the specified means of communication.
Hardware Trojan	see malicious inclusion.
liveness property	a property specifying that some desired behavior eventually occurs.
malicious inclusion	an unauthorized, undocumented modification to a piece of hardware, or hardware design unit, that circumvents or subverts some portion of the hardware's functionality
netlist	a low-level description of the connectivity of the circuits, or nets, in a hardware design.
safety property	a property specifying that some adverse behavior does not occur.
security checker	a synthesizable hardware design unit, modeling the semantics of an assertion, that is used to dynamically detect the presence of a specified prohibited behavior.
side channel attack	an attack using some physical property, such as heat, electricity, or electromagnetism, external to the functional logic of a hardware system, in order to gain information about the system.
unused circuit identification	a hardware design analysis technique that employs test stimuli on a hardware module, and marks as suspicious those circuits that are rarely activated, or not activated at all, during the test.
verification	the process of demonstrating that the requirements of a system are met by a particular implementation.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

What does it mean if we say a processor design is *secure*? Security in computing systems is often measured relative to some specific property, such as *information flow*, or a *safety* or *liveness* property, but the application of these principles to date has focused primarily on software systems. In this research, we explore the formal expression of security requirements in *hardware* designs, so that well-defined hardware security requirements and behavioral restrictions can be specified and enforced.

The principal threats to hardware design demonstrated in the academic literature to date involve some type of subversion, often called a "Hardware Trojan" or "malicious inclusion." Detecting these has proved very difficult. Most detection strategies so far employ some form of equivalence checking—equivalence of two physical samples or of two functional designs. We assert that, since the reference sample (against which equivalence is being checked) may *also* contain a malicious inclusion, equivalence itself is only sufficient to ensure the security of a design to the degree that the reference artifact is pristine. Instead, we argue that the security of a hardware design, just as in a software design, should be judged relative to well-defined policies, properties, and requirements, rather than by equivalence or non equivalence alone.

Our approach to identifying hardware malicious inclusions is based on the following observations:

- A security policy describes behaviors that are either *permitted* or *prohibited*.
- Security policies in software typically involve high-level constructs like *subject*, *object*, and *security label,* which exist at the software level of abstraction. Expressions of security policies in hardware, on the other hand, will necessarily involve lower-level constructs, which might be called *behaviors*.
- Hardware engineers already use assertions to establish the functional correctness of hardware designs; it seems natural to also use assertions, which describe behaviors, to more generally identify permitted and prohibited behaviors occurring in hardware.
- Assertions, which often derive from temporal logic, can be converted into synthesizable checkers– hardware design units which model the evaluation of an assertion formula over time against a set of input values, and can be made part of the design units being checked.

- The design of a general purpose processor is usually based on some governing document, called an architectural specification, containing descriptions of permitted and prohibited behaviors, which can be modeled using assertions.

- In the published examples of malicious inclusions to date, they often appear to cause a processor to express behaviors that are prohibited by the architectural specification. Therefore, *the action of some MIs might be detectable at runtime as violations of synthesized assertion-checkers*, evaluating those behavioral restrictions.

In order to facilitate runtime enforcement of hardware security requirements equally well in simulation, FPGA emulation, and fabricated silicon designs, we develop the notion of *security checkers*: hardware modules that can enforce PSL-specified behavioral requirements in synthesizable designs. We show how behavioral requirements stated in a text can be expressed in The Property Specification Language (PSL). Based on recently published algorithms, we create a software tool for automatically converting PSL formulas into equivalent, synthesizable hardware design entities, which can then be added into a processor design, to verify the processor's behavior. We illustrate how PSL assertions can be mapped from a processor's *architectural specification* to the design units in a specific implementation. We demonstrate, using the OpenRISC processor design, how to detect typical malicious inclusions in a processor at runtime, using the method outlined. We discuss how to apply the security checker-based methodology in simulation, FPGA emulation, and fabricated designs, including both traditional and three-dimensional integrated circuits. We explore the algorithmic complexity of our checker-generator method, and give arguments for its soundness and completeness.

Although there are theoretical limits on which security properties can be dynamically enforced using the techniques we describe, our research does provide a novel method for expressing and enforcing security requirements at runtime in hardware designs. While the method does not guarantee the detection of all possible malicious inclusions in a given processor, it addresses a large class of inclusions—those detectable as violations of behavioral restrictions in the architectural specification—which provides significant progress against the general case, given a suitably complete set of checkers.

ACKNOWLEDGMENTS

This research was funded in part by National Science Foundation Grant CNS-0910734.

The author would like to acknowledge the support and assistance of the following individuals:

- Rainer Findenig, for getting me started on checker generators,
- Matt Hicks, for sharing his thoughts and insight on malicious inclusions, and
- Cindy Eisner, for making the IBM Sugar Parser available, her excellent book with Dana Fisman on PSL, and for clarifying some of the subtleties of PSL's formal semantics.

Thanks to our collaborators in the 3Dsec group from UCSD and UCSB, including Tim Sherwood, Ryan Kastner, Jonathan Valamehr, Mohit Tiwari, and Jason Oberg, for exploring lots of ideas with us, over the phone and during our visits.

Thank you to my fellow Computer Science Ph.D. students, for many hours of thoughtful discussion on a wide variety of topics, exam preparation, notes, feedback, some grains of salt, and the occasional happy hour.

Thank you very much to all the members of my Dissertation Committee, whose broad experience and feedback have been very helpful. Thank you for your insight, guidance, and patience.

Special thanks to my advisor, Professor Ted Huffmire, for many hours of discussion, feedback, interaction and consultation with colleagues in the field, and for his patience and wisdom.

Last but not least, my deepest thanks and love to my beautiful wife, Lael, for her infinite support and understanding during this portion of our journey.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION AND PROBLEM STATEMENT

A. DISSERTATION STATEMENT

Hardware malicious inclusions in microprocessors present an increasing threat to U.S. high assurance computing systems, particularly those of the Department of Defense, due to vulnerabilities at several stages in the acquisition chain. Existing testing techniques are limited in their ability to detect these maliciously modified integrated circuits [1], [2], [3].

Hypothesis: Some processor malicious inclusions exhibit behaviors that violate restrictions contained in the processor's architectural specification, and other governing documents. Therefore, by formally expressing these restrictions using hardware assertions, and evaluating those assertions at runtime against a particular implementation, it is possible to detect some malicious inclusions in the processor implementation being evaluated. To test the hypothesis, we propose a novel method for defining and enforcing hardware security requirements, by expressing them using assertions in the Property Specification Language, and converting the assertions into a hardware-synthesizable form so they may be added to a design under evaluation, and thus identify malicious inclusions dynamically, as assertion failures.

B. RESEARCH GOALS

Our premise is that a processor architecture should identify any prohibited behaviors and define a set of behavioral restrictions which, if obeyed in an implementation, will preclude the effective operation of any related malicious inclusions. With that in mind, the general goal of this research is to improve our ability to characterize and enforce such behavioral restrictions, or security requirements, in general-purpose processors. The specific goals of this research are to:

- Examine and categorize previously demonstrated processor Malicious Inclusions.
- Develop a methodology whereby the stated security requirements of a processor architecture can be translated into runtime enforcement mechanisms that are integrated into the processor implementation, as a form of processor Execution Monitor.
- Develop tools for automating the translation from a specified security requirement into an enforcement mechanism.
- Demonstrate the methodology on a general-purpose processor model, including detection of malicious inclusions that are similar to the types demonstrated in real-world and academic examples.
- Describe how the methodology can be implemented in simulation, emulation, and in traditional and three-dimensional fabricated chips.
- Characterize what kind of security policies, in general, can and cannot be enforced at runtime in a processor Execution Monitor.
- Characterize the method's algorithmic complexity, and analyze its soundness and completeness.

1. Motivating Questions

During our investigation, we hope to answer the following questions:

1. How can we characterize the expected security threat to general-purpose processors?
2. What does it mean to say that a hardware design is *secure*?
3. How does hardware security differ from software security?
4. Against what standard can the security of a processor implementation be judged?
5. What techniques currently exist for examining the security of a processor, and what are their strengths and limitations?
6. In what manner might we formulate and express hardware security requirements, so that they can be verified to hold or not hold in a particular implementation?
7. By what mechanism can we perform such evaluation dynamically, in real time?
8. Is there a method by which we can consistently test the same security requirements across the hardware development lifecycle?
9. Is it possible to detect hardware malicious inclusions by observing violations of behavioral requirements in a processor?
10. If we implement runtime checkers for dynamic evaluation of security requirements in hardware, can we do so efficiently, and are there cases where the overhead cost may be excessive? Also, is the method sound and complete? Are there cases it will not cover?

2. Contributions

We demonstrate the following contributions as a result of this research:

- A summary analysis of the processor malicious inclusion examples published to date.
- A novel process for formalizing security requirements, in processor designs, which derive from the behavioral requirements stated in an architectural specification. We are not aware of any other hardware security method by which the security of a particular processor *implementation* is specified in terms of a set of *architectural* requirements, which are expressed in a way that allows them to be dynamically evaluated in the implementation.
- A new method for dynamically enforcing processor security requirements that is effective across nearly all phases of design and implementation, from high-level design all the way to fielded operation. We are not aware of any other hardware security method by which the same stated behavioral requirements for a processor are enforceable in simulation, in FPGA emulation, and in fabricated processor samples.
- A technique for using assertion-checkers and code coverage simultaneously, in a complementary manner, to search for malicious inclusions during high-level simulation. Previous techniques used checkers or coverage in isolation.

- A demonstration, in a real general-purpose processor design, of how the method can be used to detect some, although not all, malicious inclusions–specifically, those which manifest as a violation of behavioral restrictions in the architectural specification.
- Creation of the most complete public-domain software tool for generating synthesizable runtime enforcement mechanisms in hardware, based on temporal logic specifications. The other publicly available checker generator, *synpsl*, covers only a portion of the PSL Simple Subset, outputs only VHDL, does not provide PSL abstract syntax trees, and does not implement DFA minimization [4].[1] The two most advanced checker generators described in the literature are FoCs and MBAC, which are not publicly available in source code [5], [6]. Our checker generator is public domain, covers the PSL Simple Subset, outputs VHDL or Verilog, provides PSL abstract syntax trees, and implements full DFA minimization, as well as some boolean simplifications that do not appear to be implemented in the other tools (See Table 13).
- A description of the algorithmic complexity for each step in the checker-generator method.
- A detailed analysis of the soundness and completeness of the automata-based checker-generator method, with respect to the PSL formal semantics.

3. Areas Not Included in Scope, Not Claimed as Contributions

This research is not intended to address the following:

- Improving software security through the support of hardware-based security features.
- The detection of malicious software, or software subversions in operating systems.
- Attacks on processors that involve physical side channels, such as electromagnetic phenomena.
- Methods for automatically determining what the security requirements of a given hardware application *ought* to be; these are application-dependent, and will vary according to the intent of the architect, who may elect to impose a small number of behavioral restrictions, or none at all.

We do not claim the following contributions with this research:

- A method for detecting any and all conceivable processor malicious inclusions.
- A method for enforcing any expressible security policy requirement in a processor.

[1] We obtained the source code for this tool from the author, via a Creative Commons license.

THIS PAGE INTENTIONALLY LEFT BLANK

II. THE THREAT TO HIGH ASSURANCE SYSTEMS

> "Due to cost-cutting pressures the design and manufacture of the majority of [integrated circuits] and other components are outsourced to third parties overseas. In particular, it is expected that, by the end of this decade, the majority of integrated circuits will be fabricated in cheap foundries in China."
>
> –Tehranipoor and Sunar (Sadeghi, "Towards Hardware Intrinsic Security," 2010)

A. THE NEED FOR PROCESSORS IN HIGH ASSURANCE SYSTEMS

Today's Defense Department relies on advanced microprocessors for its high assurance needs. Those applications include everything from advanced weaponry, fighter jets, ships, and tanks, to satellites and desktop computers for classified systems. Much attention and resources have been devoted to securing the software that runs these devices and the networks on which they communicate. However, two significant trends make it increasingly important that we also focus on securing the underlying hardware that runs these high-assurance devices. The first is the United States' greater reliance on processors produced overseas. The second is the evolution in the complexity of hardware, along with the ease of making malicious changes to it.

B. THE PROCESSOR DESIGN LIFECYCLE

A general-purpose processor's life cycle spans several distinct phases, as summarized in Figure 1.

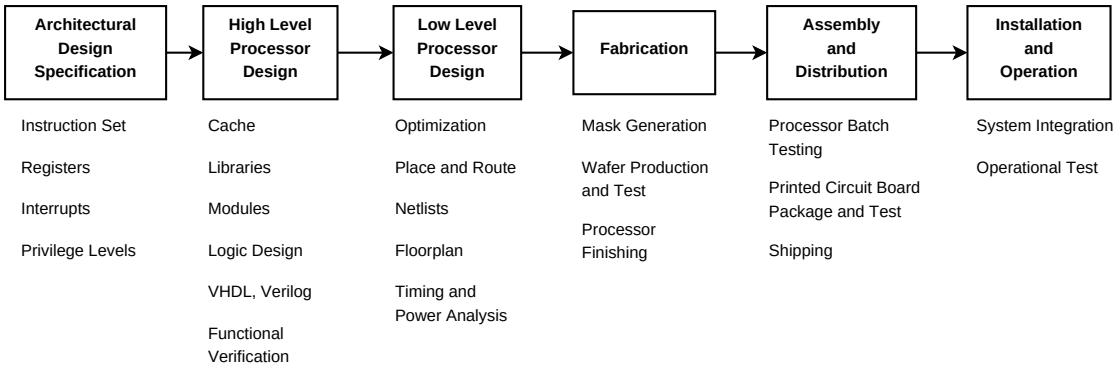


Figure 1: Processor Design Life Cycle.

The potential for malicious modification varies from stage to stage. For example, some processors are designed and verified in facilities certified to be "trusted." This does not render their designs invulnerable to attack, but gives us relatively greater confidence in their fidelity at this stage. However, the physical fabrication process, especially for high-performance processors, is largely beyond the control of the Department of Defense, for the latest-generation processor technology. At the other end of the design cycle, installation

and operation, DoD and other high-assurance users will again normally have tight control of a processor's fielded environment. The Defense Advanced Projects Research Agency (DARPA) has provided industry with a subjective assessment snapshot of the relative risk in each of the phases, as shown in Figure 2 [7].

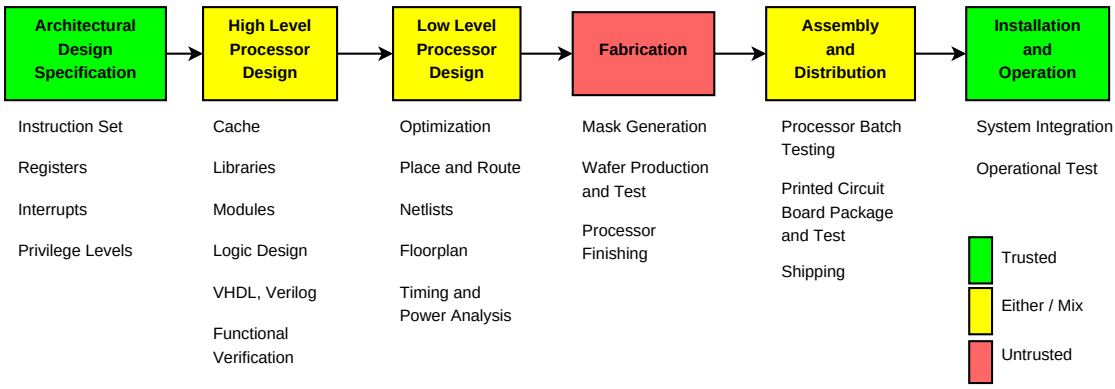


Figure 2: Processor Design Life Cycle, With DARPA Risk Assessment.

1. Design-Stage Modification

Since modern processors are designed in software, the processor design plans become a potential target of attack. John Randall, a semiconductor expert at Zyvex Corp., notes that "any malefactor who can penetrate government security can find out what chips are being ordered by the Department of Defense and then target them for sabotage. If they can access the chip designs and add the modifications, then the chips could be manufactured correctly anywhere and still contain the unwanted circuitry" [8].

Even the design tools themselves can be subverted. Roy, Koushanfar, and Markov demonstrated that maliciously modified CAD software can be used to plant MIs in integrated circuits [9].

2. Post-Fab Modification

Such undetected logic can even be inserted after a chip has been manufactured.

> Chip alteration can even be done after the device has been manufactured and packaged, provided the design data are available, notes Chad Rue, a [processor] engineer . . . Skilled circuit editing requires electrical engineering know-how, the blueprints of the chip, and an etching machine [which] shoots a stream of ions at precise areas on the chip, mechanically milling away tiny amounts of material . . . You can remove material, cut a metal line, and make new connections . . . The results can be astonishing: a knowledgeable technician can edit the chip's design just as easily as if he were taking 'an eraser and a pencil to it.' [8]

C. SUPPLY CHAIN VULNERABILITIES

Every year, more microprocessors destined for U.S. DoD systems are manufactured overseas, and fewer are made inside the United States. As more processors are manufactured in countries whose interests

are not aligned with those of the United States, there is a greater risk of processors being manufactured with malicious inclusions, which could compromise high-assurance systems. This concern was highlighted in a 2005 report by the Defense Science Board, which noted a continued exodus of high-technology fabrication facilities from the United States [10]. Since this report, "more U.S. companies have shifted production overseas, have sold or licensed high-end capabilities to foreign entities, or have exited the business" [11]. One of the Defense Science Board report's key findings reads,

> Throughout the past ten years, the need for classified devices has been satisfied primarily through the use of government owned, government- or contractor-operated or dedicated facilities such as those operated by the The National Security Agency (NSA) and Sandia [National Laboratory]. The rapid evolution of technology has made the NSA facility obsolete or otherwise inadequate to perform this mission; the cost of continuously keeping it near to the state of the art is regarded as prohibitive. Sandia is not well suited to supply the variety and volume of DoD special circuits. There is no longer a diverse base of U.S. integrated circuit fabricators capable of meeting trusted and classified chip needs. [10]

1. Overseas Design and Fabrication

Today, most semiconductor design still occurs in the U.S., but some design centers have recently developed in Taiwan and China [12]. In addition, major U.S. corporations are moving more of their front-line fabrication operations overseas for economic reasons:

- "Cisco Systems has pronounced that it is a 'Chinese company,' and that virtually all of its products are produced under contract in factories overseas" [11].
- "Press reports indicate that Intel received up to $1 billion in incentives from the Chinese government to build its new front-end fab in Dalian, which is scheduled to begin production in 2010" [13].
- "Raising even greater alarm in the defense electronics community was the announcement by IBM to transfer its 45-nanometer bulk process Integrated Circuit (IC) technology to Semiconductor Manufacturing International Corporation (SMIC), which is headquartered in Shanghai, China. There is a concern within the defense community that it is IBM's first step to becoming a 'fab-less' semiconductor company. IBM is the only state-of-the-art IC manufacturer that has a 'trusted' take-or-pay contract with the Defense Department and the National Security Agency at its plant in Vermont. Intel, the other cutting-edge U.S. integrated circuit maker, does not want to do dedicated work for the U.S. government" [11].

Adee notes, "almost all Field Programmable Gate Arrays (FPGAs) are now made at foundries outside the U.S., about 80 percent of them in Taiwan. Defense contractors have no good way of guaranteeing that these economical chips haven't been tampered with. Building a kill switch into an FPGA could mean embedding as few as 1,000 transistors within its many hundreds of millions" [8].

2. Counterfeits

The complexity of the processor acquisition chain makes it difficult to separate genuine parts from fakes, as well. The ease with which counterfeit processors have made their way into the The Department of Defense (DoD) supply chain is illustrated in the following news reports [14]:

- From November 2007 through May 2010, U.S. Customs officials said they seized 5.6 million counterfeit chips.
- Processors ordered for an F-15 flight control computer were discovered to be counterfeit.
- Two men indicted in October, 2010, admitted importing from China more than 13,000 fake chips altered to resemble those from legitimate companies, including Intel, Atmel, Altera and National Semiconductor. Among those buying the chips was the U.S. Navy.

The counterfeiters' methods and motivations are outlined by Johnson:

> To withstand the rigors of battle, the Defense Department requires the chips it uses to have special features, such as the ability to operate at below freezing temperatures in high-flying planes. And because it pays extra for such chips, experts say, it has become a prime target for counterfeiters... Counterfeiters—many of them based in China—often tear apart scrapped computers to obtain chips, which they then mislabel to appear suitable for jobs that exceed the parts' capabilities. That can result in the components suffering dangerous glitches. [14]

If a counterfeit processor is functionally equivalent to the target being copied, but made with cheaper materials or inferior processes, then functional testing in a normal environment is not likely to detect that it is an imitation. Although fake processors may not necessarily contain malicious modifications to their functionality, their sheer volume suggests a potential avenue for introducing subverted processors into some high-assurance U.S. products.

D. THREAT SUMMARY

High performance general purpose processors used in DoD high-assurance systems are increasingly being manufactured and assembled overseas, often in countries whose interests are not clearly aligned with the interests of the United States. An adversary with sufficient resources could maliciously modify a general purpose processor at several different stages of the acquisition chain, from design and fabrication to assembly and transport. Due to the complexity of the supply chain, modified processors could well find their way into high-assurance systems.

III. MALICIOUS INCLUSION CHARACTERISTICS

The term Hardware Trojan is commonly used to describe a hardware malicious inclusion. We prefer the latter term, because a Trojan requires some action of acceptance by the victim (e.g., clicking on a link in an e-mail, or admitting the horse through the gates of Troy) whereas the victim will normally not be aware of the type of hidden subversion associated with a hardware malicious inclusion. We define a malicious inclusion (MI) as "an unauthorized, undocumented modification to a piece of hardware, or hardware design unit, that circumvents or subverts some portion of the hardware's functionality."

A. REAL-WORLD REPORTS

Though reports of actual malicious inclusions are often classified or kept quiet for other reasons, some reports do surface, as in the following examples:

- "According to a U.S. defense contractor who spoke on condition of anonymity, a 'European chip maker' recently built into its microprocessors a kill switch that could be accessed remotely. French defense contractors have used the chips in military equipment, the contractor told IEEE's Spectrum magazine. If in the future the equipment fell into hostile hands, 'the French wanted a way to disable that circuit,' he said" [8].
- According to the New York Times, such a "kill switch" may have been used in the 2007 Israeli raid on a suspected Syrian nuclear facility under construction. The Times report cites an unnamed American semiconductor industry executive, claiming direct knowledge of the operation [15].
- According to independent researchers, AMD left an undocumented, password-activated, debugging "backdoor" in a common family of processors, permitting access to certain machine status registers [16].

B. ACADEMIC DEMONSTRATIONS

Academic researchers have demonstrated a number of different malicious inclusions in recent years. Some target application-specific processors, like encryption chips, and others target general-purpose processors. A summary of some prominent examples follows.

Jin, Kupp, and Makris described their experiences designing MIs for the 2008 New York University Cyber Security Awareness Week Embedded Systems Challenge [17]. They employed eight different attacks against an FPGA implementation of the Alpha encryption module, showing various methods of leaking the encryption key and disabling the chip. See Table 1 (data from [17]).[2]

Some of the most sophisticated attacks to date were developed by King et al., targeting a Leon3 SPARC platform. The authors employed a combined hardware-software approach, in which general-purpose

[2] In the accompanying tables, for MI size, we define Small as <.1% of design area, Medium as .1-1%, and Large as > 1%, according to the data in the publications.

Processor Elements Targeted	MI Size	Trigger Type	Trigger Description	Attack Type	HW, SW, or Both	Attack Description
Keyboard I/O	Med.	Keyboard I/O (physical access)	Text input "New Haven."	Modify Function - Leak Key	HW	Output cipher text replaced by key.
Keyboard I/O	Large	Keyboard I/O (physical access)	Function key pressed.	Disable	HW	Chip disabled.
Chip Text I/O	Large	I/O to Chip	Text input "Moscow."	Modify Function - Corrupt Output	HW	Output replaced with "Moscow."
Chip Text I/O	Small	I/O to Chip	Input 1KB block of plaintext.	Modify Function - Leak Key	HW	End of ciphertext replaced by key.
Chip Text I/O, RS-232	Med.	I/O from Chip, RS-232	New key legitimately installed.	Modify Function - Leak Key	HW	New key hidden in output.
Chip I/O output	Med.	Counter Exceeds Preset Number	More than N characters sent.	Disable	HW	Chip disabled.
Chip I/O	Large	Attacker gets control of T/R port (physical access)	Attacker accesses port.	Modify Function - Leak Key	HW	Real key is encrypted using attacker's key.
Keyboard I/O	Large	Keyboard I/O (physical access)	"Caps Lock" pressed.	Modify Function - Leak Key	HW	Key flashes on keyboard LED.

Table 1: Malicious Inclusions by Jin, Kupp, and Makris.

hardware subversions were used by malware to take control of the operating system. They installed an escalation-of-privilege attack, and a "shadow mode" attack, which used a hidden register set. See Table 2 (data from [18]).

Waksman and Sethumadhavan constructed twenty-one attacks specifically targeting the execution pipeline of an OpenSPARC processor platform, and implemented approximately half of them [19]. Most of the attacks employed access control violations during some form of *load* or *store* operation.

C. CHARACTERISTICS

Several researchers have described the characteristics of malicious inclusions, or hardware Trojans, in taxonomy form. Wang et al., gave the first detailed taxonomy in 2008 [20]. Similar versions have since been presented by Rajendran et al. [21], and Karri et al. [22] (see Figure 3). Karri et al. categorize malicious inclusions according to five different features: Insertion Phase, Abstraction Level, Activation Mechanism, Effects, and Location [22]. Their categorization is summarized in the following sections.

Processor Elements Targeted	MI Size	Trigger Type	Trigger Description	Attack Type	HW, SW, or Both	Attack Description
MMU, Data Cache	Small	Software Instruction	Byte sequence seen on data bus.	Modify Function	Both	"Memory Access." A sequence of bytes on the data bus causes the MMU to ignore CPU privilege levels for memory access. Implemented by modifying the data cache. Privilege escalation attack.
L1 cache	Small	Network I/O	UDP packet arrives with trigger sequence.	Modify Function	Both	"Shadow Mode". Adds a new processor mode, similar to an ISA extension. Shadow-mode instructions have full processor privileges. Reserves special instruction cache lines and data cache lines for the attack. Password sniffer and login backdoor service.

Table 2: Malicious inclusions by King et al.

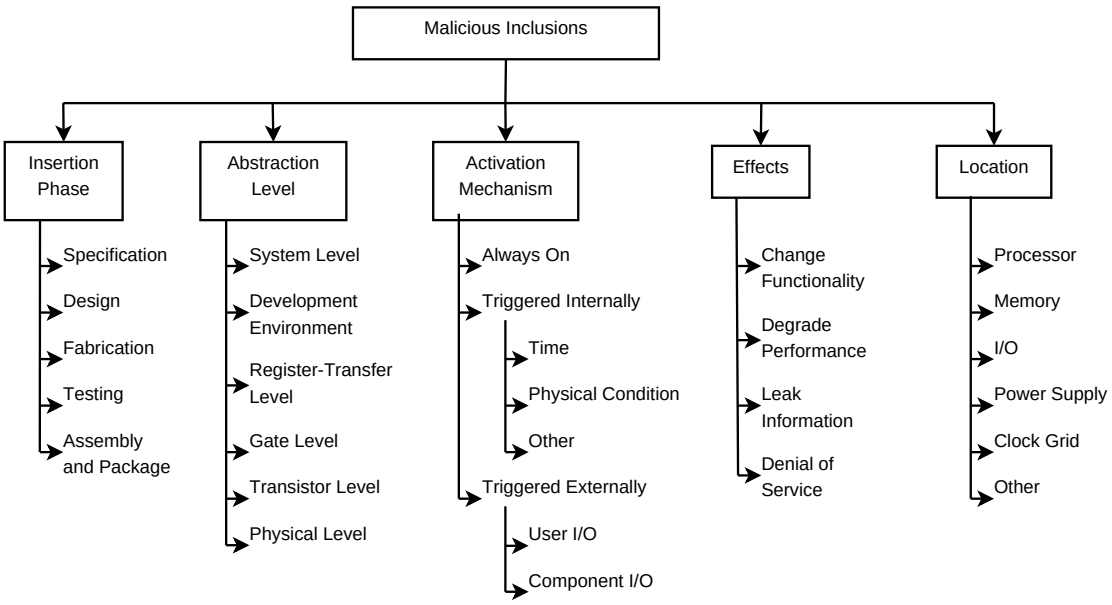


Figure 3: Malicious inclusion taxonomy.

- *Insertion Phase*: When in the hardware design lifecycle is the MI added?
  - Specification: Chip designers describe the system's functional characteristics.
  - Design: Developers map out the functional, logical, physical, and timing constraints, and possibly incorporate Third-Party Intellectual Property (3PIP).
  - Fabrication: Includes mask creation, layered chemical processes, and wafer and die production.
  - Testing: An attacker can insert flaws here, or design them to be invisible to known testing procedures.

    - Assembly: Components are assembled onto a Printed Circuit Board (PCB). A subversion anywhere in the assembled system can compromise its entirety.

- *Abstraction Level*: At what design fidelity is the MI deployed?

    - System Level: Hardware modules, interconnect, and intermodule communication are defined.
    - Development Environment: Includes Computer-Aided Design (CAD) tools, synthesis tools, and automated scripts.
    - Register-Transfer Level: Where developers define the hardware entities in terms of named signals, storage units, and functional logic.
    - Gate Level: The level of abstraction where the design is synthesized into fundamental logic gates (AND, OR, NOT, DFF, etc.) that can be floorplanned and manufactured.
    - Transistor Level: Includes individual transistors and their power and timing characteristics.
    - Physical Level: Defines the full physical structure of a processor, such as the physical layout of the transistor elements, metal interconnects, and structural and non-conducting layers.

- *Activation Mechanism*: What causes the MI to begin working?

    - Always On: The MI has no trigger; it is constantly activated.
    - Internally Triggered: Such as by a counter or a physical condition.
    - Externally Triggered: As by user I/O, through a component's data stream.

- *Effects*: What does the MI do?

    - Change Functionality: Cause a unit to behave in a manner not in accordance with its specification, possibly including the corruption of data.
    - Degrade Performance: Cause a unit to function sub-optimally, for example by slowing down or consuming more power.
    - Leak Information: Effect the extraction of information through unintended means.
    - Deny Service: Temporarily or permanently disable, or even destroy, the targeted system.

- *Location*: Where in the hardware design is the MI placed?

    - Examples: Processor control, memory, I/O, power supply, debug circuits, virtualization management, BIOS, etc.

D. SOME STATISTICS

Karri et al. compiled MI statistics from an accumulation of examples at the 2008 Embedded Systems Challenge, mentioned earlier, along with data from previously published MIs [22]. We summarize the compilation in Figures 4 through 6. Note the statistics may be somewhat biased from the full spectrum of real-world attacks because:

- Most of these attacks were against one model of encryption unit on an FPGA.
- For the attacks at the Embedded Systems Challenge, contestants were required to insert one or more Trojans that leak a key, leak text, or create a denial of service.

Nevertheless, the summary charts give a subjective indication of some aspects of how real-world Trojans might be designed.

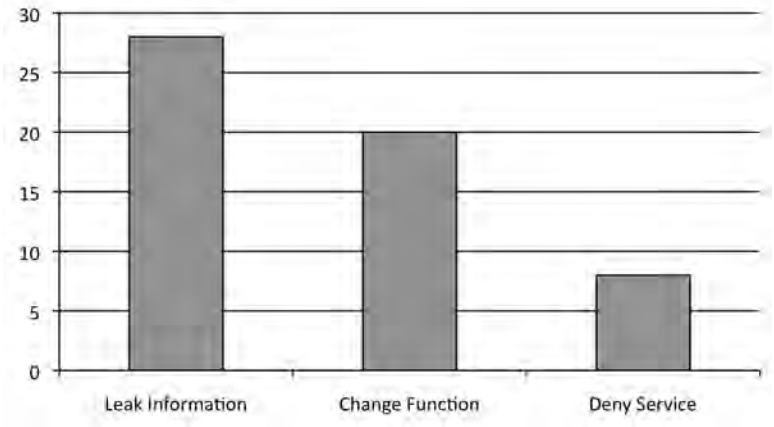


Figure 4: Number of MIs, by functional type.

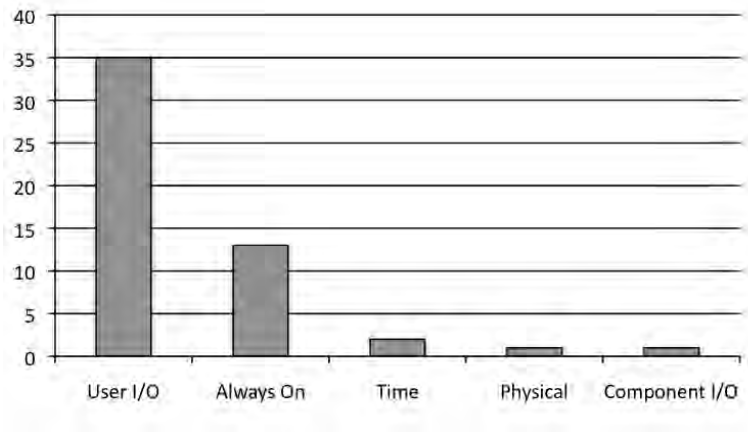


Figure 5: Number of MIs, by trigger type.

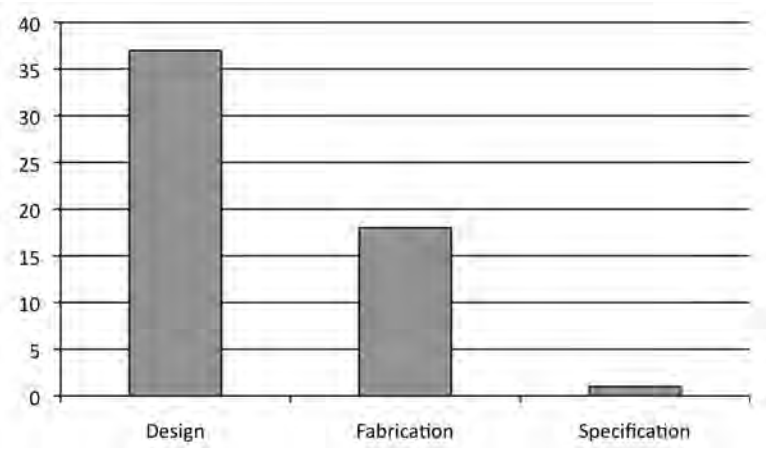


Figure 6: Number of MIs, by design phase in which they were inserted.

E. THE PROCESSOR THREAT MODEL

In theory, a malicious modification to a processor's design could take almost any size and form, and attack almost any circuit in a processor; in practice, though, malicious inclusion design will be governed by several limiting factors:

- The larger the modification, the easier it will be to detect through physical analysis. Researchers have successfully demonstrated nondestructive physical detection of some modifications occupying as little as approximately .01% of the total processor area [1], [23].
- Malicious modifications to a processor's high-level design could be uncovered as errors during functional verification, and therefore will likely be constructed specifically to *avoid* causing any failures in functional verification.

Therefore, we are concerned with detecting malicious inclusions that are relatively small in size, do not violate functional verification tests, and target circuits whose function is related to the common subversion types suggested by the earlier statistical analysis. We assume that the adversary's primary goals will depend on the hardware-hosted application, but are likely to center on either extraction of information, denial of service, and functional modification, as seen in the examples. We also note that, due to the presence of on-off triggers and delayed activation [3], we can make no *a priori* assumptions about *when* a malicious inclusion will be active or inactive.

F. SUMMARY

Hardware malicious inclusions have been demonstrated by a variety of academic researchers, and discovered in a few real-world examples. Though the sample size of MIs demonstrated to date is not large, we can observe from this review a few common themes that may be useful in guiding our security efforts.

- Motive: in general, the most common motives appear to be:

    - Theft of some data
    - Change of a target function
    - Some form of disablement
- Means: common MI techniques employed include:
    - Unauthorized Escalation of some inherent “privilege” within the processor
    - Bypass of some internal processor access controls
    - Direct exfiltration of data through other than normal means

Although this is not an exhaustive list, security requirements for a hardware design should include assurances against these key attack vectors, at a minimum.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. SECURITY POLICIES AND PROCESSORS

"A security policy defines execution that, for one reason or another, has been deemed unacceptable."

–Fred Schneider, "Enforceable Security Policies," 2000.

A. LEVELS OF ABSTRACTION

In any computing system, different concepts exist at different levels of abstraction. High-level software constructs, such as those found in object-oriented programming, may or may not be meaningful to the operating system kernel. Low-level software constructs, such as a small loop performing an iterative computation over an array, are expressed at a finer granularity than object-oriented concepts, or the *subjects* and *objects* defined in an operating system security policy, for example. Similarly, concepts defined in software, like *types* [24], may not have meaning down at the processor level, in hardware. A passage from Hamming's book "Learning to Learn: The Art of Doing Science and Engineering" [25] emphasizes this point:

> We see that the machine does not know where it has been, nor where it is going to go; it has at best only a myopic view of simply repeating the same cycle endlessly. Below this level, the individual gates and two-way storage devices do not know any meaning—they simply react to what they are supposed to do. They too have no global knowledge of what is going on, nor any meaning to attach to any bit, whether storage or gating . . . it is we who attach meaning to the bits [emphasis in original]. The machine is a machine in the classical sense; it does what it does and nothing else.

A similar observation comes from hardware security researchers Waksman and Sethumadhavan [19]:

> Because hardware components (including backdoors) are architecturally positioned at the lowest layer of a computational device, it is very difficult to detect attacks launched or assisted by those components: it is theoretically impossible to do so at a higher layer, e.g., at the operating system or application [level], and there is little functionality available in current processors and motherboards to detect such misbehavior. The state of practice is to ensure that hardware comes from a trusted source and is maintained by trusted personnel—a virtual impossibility given the current design and manufacturing realities. In fact, our inability to catch accidental bugs with traditional design and verification procedures, even in high-volume processors, makes it unlikely that hidden backdoors will be caught using the same procedures, as this is an even more challenging task.

B. TRADITIONAL SECURITY POLICIES DESCRIBE SOFTWARE-LEVEL ENTITIES

In light of the previous discussion, we mention a few popular security models and concepts, and some of their characteristic constructs, in order to show that they necessarily exist above the hardware level of abstraction:

- Basic access control policies: *Subjects, Objects, Authorizations* [26]
- Lattice-Based Information Flow Policy: *Objects, Subjects, Security Classes* [27]
- Noninterference-Based Information Flow Policy: *Users, States, Commands, Outputs* [28]
- Integrity Policy: *Users, Constrained Data Items, Transformation Procedures* [29]
- Reference Monitor Concept: *Subjects and Objects* [30]
- Covert Channel Analysis: *Subjects and Shared Objects*[3] [31]

In each case, at least one of the constructs on which the policy is defined, such as *subject* or *object*, is defined at the *software* level of abstraction. Though the constructs in a processor, such as a memory word, an interrupt, or an executing instruction, may support one of these higher-level constructs, the processor has no built-in awareness of what is represented by them at the higher level.[4] Since the traditional types of security policies listed above are based on entities at the software level of abstraction, we will have to employ different methods for describing security properties solely in hardware. Our investigation focuses on detecting malicious subversions in a processor, deriving our security requirements from the behavioral restrictions listed in the processor's *architectural specification*.

C. SIDE CHANNELS

It is important to differentiate *covert channel* attacks from *side channel* attacks. Side channel attacks use some property, often a physical property such as heat or electricity, external to the logic of a hardware system, in order to gain information about the system. An example is externally evaluating the electromagnetic characteristics of a circuit while the circuit performs cryptographic computations, in order to deduce some properties of the unencrypted data or the encryption key. Side channel attacks and analysis are important in hardware security, but are not within the scope of this investigation.

D. SUMMARY

Because higher-level software constructs like *subject* and *object* are abstracted away as we move down to the hardware layer, we will attempt to describe hardware security requirements at a *lower* level of abstraction than software security policies, like those mentioned above. In Chapters VI and VII, we will explore the use of lower-level forms of expression, such as *properties*, as expressed by *assertions*, which can, in fact, be specified and enforced at the hardware level, unlike the higher-level *policies* mentioned in Section B. Although they describe behavior at a lower level, suitable for hardware, these more basic constructs can still be used to characterize *permitted* and *prohibited* behaviors in a system [34].

[3] A covert channel is a conduit through which information can be conveyed from one subject (or process acting on its behalf) to another subject, via a shared object, or "shared resource attribute." Covert channel analysis, though not a formally-specified security policy, is a key security technique for identifying unintended information flows. It is important to observe that, while the processes that communicate with each other via a covert channel exist in software, the "shared resource attribute" can exist at either the software or the hardware level. Because a complete covert channel analysis in a hardware-software system requires concurrent analysis of the system *software*, we do not include it explicitly in this research, which focuses specifically on the *hardware* portion.

[4] Though some research has been done on using hardware support mechanisms to help facilitate the enforcement of security policies at the software level [32], [33], those methods are independent of the ones explored here.

V. RELATED WORK

Verification of a hardware design can employ both static and dynamic methods; our research focuses on the latter. In this chapter, we first describe both static and dynamic analysis in the *functional* verification of hardware designs, and how our analysis method compares to them. Then, we discuss some existing methods for analyzing the *security* of a hardware design, to illustrate some of the motivation for our method.

A. STATIC AND DYNAMIC ANALYSIS OF HARDWARE DESIGNS

1. Static Analysis

a. Theorem Provers

Static verification is normally associated with *formal methods*, such as using *theorem provers*, like PVS, ACL2, and HOL. Some PSL assertions have even been modeled in PVS and HOL [35]. One example of functional verification using formal methods is the work of Centaur Technology. That group, including Slobodová, Davis, Swords, and Hunt, used an ACL2 framework to formally verify the correctness of mathematical operations in an x86 processor called the Via Nano [36].

One advantage of proving properties of a design statically is that there is no need to generate an input stimulus (testbench) for the design. Another advantage is that, once a proof (or counterexample) is arrived at, a property may be proven to hold (or not hold), with certainty. A disadvantage of static analysis is that its computational complexity can be prohibitive, and some properties may be statically undecidable [6], [37].

b. Model Checkers

Model checking is another common form of static verification, for finite state systems. In general, a model checker examines the possible states of a system, and determines whether or not certain defined properties are true in all states. Popular model-checking tools for hardware include SMV, IBM RuleBase, and Cadence SMV. For example, RuleBase was used by Geist, Landver, and Singer to verify a processor bus interface [38], and by Goel and Lee to verify a bus arbiter [39]. Similarly, Parash used RuleBase to verify the functionality of an MPEG-2 decoder [40], and Chavet used it to verify a SHA-1 hashing circuit [41]. In another example, Patankar, Jain, and Bryant used symbolic trajectory evaluation, a form of model checking, to verify the correctness of machine instructions in an implementation of an ARM-variant processor [42].

Model checkers do suffer from "state-space explosion" as they consider longer input sequences and more complex designs, however. They are therefore often more efficient in finding counterexamples to a property, rather than demonstrating that a property holds in all possible states of the system [43].

2. Dynamic Analysis

Dynamic verification of a hardware design, on the other hand, requires the tester to construct a set of input stimuli, e.g., a testbench, to exercise the hardware modules. Covering all test cases, especially in a large

design, can be very challenging—the difficulty in tractably generating all possible input stimuli means that a property which passes dynamic verification may not be proved with certainty to always hold [6]. However, dynamic verification is often simpler to perform than formal static analysis. According to Boulé and Zilic, "Dynamic verification is the predominant verification approach used in practice, and is most often associated with simulation" [6].

There are numerous commercial products for dynamic evaluation of hardware designs, in particular for assertion-based verification (ABV). For example, Mentor's ModelSim and Synopsis' VCS are popular products that support ABV. Both PSL assertions and SystemVerilog Assertions (SVA) are in common commercial use, and so are assertion-based frameworks, like OVM [44].

3. Static and Dynamic Assertion-Based Verification

Assertion-based verification is simply the use of assertions to verify whether properties hold true in a particular design. In hardware, the assertions may be specified, for example, using SVA or PSL. Assertions may be used to support either static or dynamic verification, though in practice they are more often associated with the latter. As an example of static verification based on assertions, Tuerk, Schneider, and Gordon demonstrated a model checking and theorem proving infrastructure for a subset of PSL [45].

Until recently, dynamic assertion-based verification meant only simulation. An important development in the use of assertion-based methods for dynamic hardware analysis is the ability to construct *assertion checkers*, which are synthesizable hardware design units that can check the status of temporal-logic assertions over time, against a set of inputs. By creating synthesizable assertion checkers, we can now move assertion-based dynamic verification beyond just simulation, and apply it in FPGA emulation and fabricated designs, as well, as discussed in Chapter VII. Techniques for constructing synthesizable assertion checkers have been developed by Boulé and Zilic, among others [5], [6], [46], and are central to our research.

A cursory taxonomy of assertion-based verification techniques is shown in Figure 7.

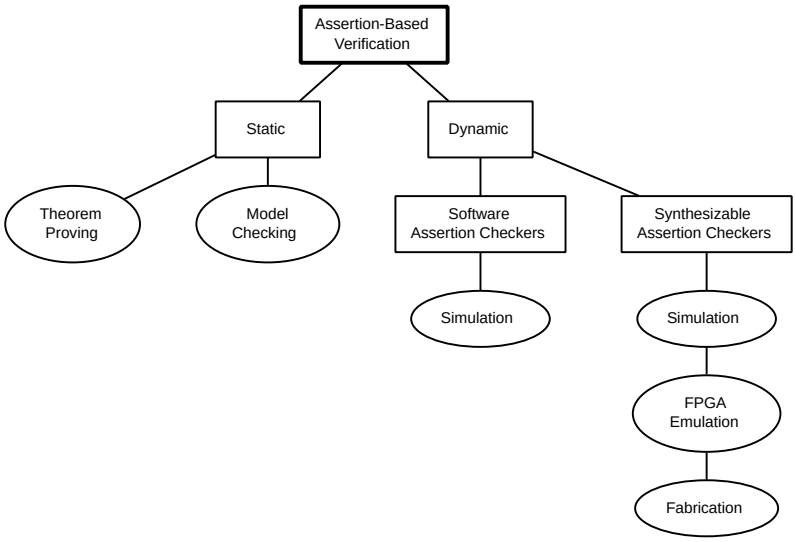


Figure 7: Assertion-based verification techniques.

4. Conclusion

Because we are interested in mechanisms which can be used in testing and also remain part of a fielded, operational piece of hardware, we focus our research on dynamic, or runtime, verification methods, rather than static methods. However, at the high-level design stage, static and dynamic methods are complementary, and can be applied independently to the same processor model.

In Chapter VII, we describe our own assertion-based dynamic analysis method in detail; for context, we compare it here to established functional verification methods, in Table 3. The method we develop employs assertion checkers, listed in the bottom row. In general, functional verification methods will try to *confirm positive behaviors*, or verify adherence to some specification; in contrast, methods like our assertion checker-based technique, for detecting malicious inclusions, will primarily try to *detect negative behaviors*, or violations of some restriction. We explore this distinction further in Chapter VII.

Method	**Static or Dynamic**	**Purpose**	**Description**	**Tools**
Model Checkers	Static	Functional Verification	State-Space Exploration	RuleBase, VIS, SMV
Theorem Provers	Static	Functional Verification	Formal Proofs	PVS, ACL2, HOL
Assertion-Based Verification	Dynamic	Functional Verification	Simulation of Assertions	SVA, PSL, simulators
Hardware Assertion Checkers	Dynamic	Functional Verification *or* MI Detection	Synthesizable Checkers	PSL, psl2hdl

Table 3: Comparison of static and dynamic hardware analysis methods.

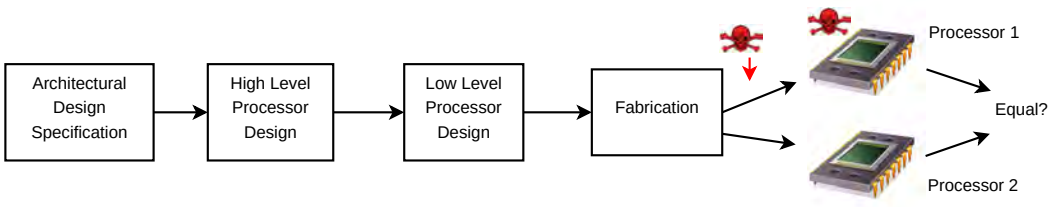
In the next section, we set aside the closely-related topic of *functional* verification, and discuss several existing methods for analyzing a processor, or a hardware design, for malicious inclusions.

B. EXISTING HARDWARE SECURITY METHODS

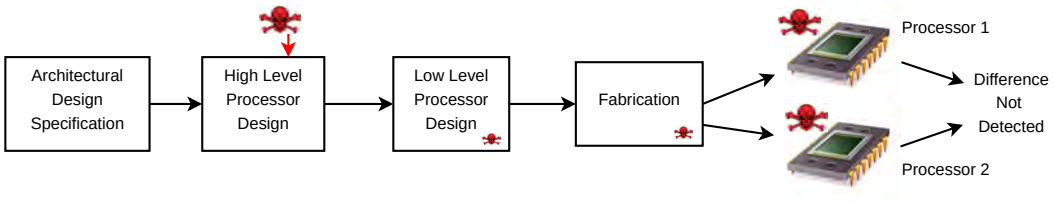
The current state of the art for manufacturing trustworthy processors generally follows one of two approaches, according to Tehranipoor and Suna [47]:

> The first option is to make the entire fabrication process trusted. This option is prohibitively expensive with the current trends in the global distribution of the steps in IC design and fabrication. The second option is to verify the trustworthiness of the manufactured chips upon returning to the clients. To achieve this, it would be necessary to define a post-manufacturing step to validate conformance of the chip with the original functional and performance specifications.

Existing methods for detecting malicious inclusions are primarily based on the detection of physical changes in the power and timing characteristics of a processor, as observed from its input and output ports. These techniques rely on possession of a known good, or "golden," sample processor, which acts as a baseline, against which other processors are judged. This method may detect changes made to a processor in the design or fabrication stage. However, it will not detect an earlier, high-level design change that makes its way into all the processors in a production run, since the "golden" sample would also be affected. The difference is shown in Figure 8. In example (a), an MI is inserted during fabrication, and the physical difference from another processor is detected; in example (b), the MI is inserted into the high-level design, it is fabricated into both processors, and no physical difference is detected.



(a)



(b)

Figure 8: Limitation of physical equivalence checking.

In contrast, to detect malicious inclusions inserted during development, we focus on comparing the behavioral restrictions, from the architectural specification, with the runtime behavior of the implemented system.

1. Physical Analysis of Processors

A physical processor may be analyzed for the presence of malicious inclusions by methods that are either destructive, in which the sample is destroyed, or methods that are nondestructive, in which the sample may still be used.

a. Destructive Methods

It is possible to deconstruct a physical sample processor for analysis. The basic method involves backside thinning, using a combination of chemical removal and mechanical polishing, to expose the metallization layers and transistors. Once exposed, the physical structure of the interconnects and transistors can be imaged. With the physical structure mapped, it is possible to reverse engineer a sample's processor mask, and compare it with a reference processor mask set to identify deviations. One such effort is being conducted on behalf of DARPA [48].

There are several challenges associated with this type of analysis:

- Imaging a processor's features is difficult due to the resolution required. Modern feature sizes are less than 50 nanometers, pushing the resolution limit of all but the most sophisticated microscopes.
- The imaging is relatively time-consuming, given the large number of structures that must be imaged.
- Not all features are easily imaged by one technology; for example, X-ray imaging may pick up the metal layers, but may fail to distinguish non-metal features in the transistors.
- The technique is necessarily destructive, and too involved to be applied to more than a small sample of processors.
- The reference processor mask may be corrupted.

b. Nondestructive Methods

It is also possible to check for the equivalence of two processors using only their external interfaces. According to Tehranipoor and Koushanfar, MIs "typically change a design's [externally observable] parametric characteristics—for example, by degrading performance, changing power characteristics, or introducing reliability problems in a chip" [3]. By precisely measuring and comparing the power consumption and timing data of two processors, one can use statistical analysis to infer the presence or absence of modified circuitry, compared to the reference unit [49], [20]. The principal limitations of these methods are:

- They only verify equivalence; if the trusted reference design and the device under test have both been subverted, then the subversion will not be detected.
- MIs representing a change of less than approximately .01% of the overall chip area are not likely to be detected using methods published to date [1], but effective MIs can be constructed that are very small in terms of their relative size [18].

The physical analysis limitations described above suggest the need for other MI detection techniques.

2. Design Analysis of Processors

In addition to the methods mentioned by Tehranipoor and Suna, there is another, emerging category of hardware trust methods, which we refer to collectively as *design analysis* methods. In a design-analysis method, MIs are detected through behavioral and functional analysis of a high-level processor design, usually prior to fabrication. The method we propose in Chapter VII falls into this category.[5] There are also other design analysis techniques proposed by Banga and Hsiao, Hicks et al., and others, discussed next.

a. Functional Equivalence

In addition to physical equivalence-checking, it is also possible to check the equivalence of two processor implementations while they are still in the design stage, before fabrication. For example, we may want to compare the functional equivalence of a high-level RTL design with its synthesized low-level counterpart, a combined netlist,[6] or compare two high-level designs with each other. If some subversion has been introduced in the circuit under test, it should be detectable as a functional difference. One potential complication in comparing high-level and low-level designs is the need to account for the optimization process, during which some internal signals may be trimmed away, while a module's inputs and outputs remain the same.

Banga and Hsiao proposed an MI-detection method called Trusted RTL, which relies in part on equivalence-checking between a reference circuit design and a circuit under test [50]. Trusted RTL first trims out "easily" activated circuits, considering them not likely to be malicious, and then compares the leftover portions of the design, between the reference circuit and circuit under test.

In an ongoing DARPA-sponsored research initiative called Iris, researchers seek to evaluate functional equivalence, but in a different way. The idea behind Iris is to develop methods of reconstructing, from a low-level format like a netlist, a design's high-level functionality. Once reconstructed, it can be compared with a high-level reference design for malicious modifications [51].

b. Unused and Rarely-Used Circuits

Several researchers have proposed identifying unused or rarely-used circuits in a design as likely MIs, similar to the method applied in Trusted RTL.

Hicks et al. outlined a technique for detecting some malicious design-stage modifications [52]. In their approach, called Blue Chip, the high-level design is analyzed for potential malicious inclusions by exercising the processor testbench;[7] those circuits not exercised by the testbench are presumed suspicious, and removed from the design. In the Blue Chip approach, the malicious change must already be present in the high-level design; if it is introduced afterward, Blue Chip will not detect it. The approach also relies on the correctness and thoroughness of the processor testbench.

[5] The method we propose can also be used in FPGAs and fabricated designs, as well, depending on customer requirements, but is easiest to perform during simulation. It begins with design analysis, but can be extended to dynamic runtime checking in physical systems.

[6] A netlist is a file format used for gate-level representation of a hardware design. The individual circuits are often called "nets."

[7] A testbench is a separate hardware design unit which provides input, or stimuli, to a hardware design unit, such as a processor, for evaluation purposes.

One potential pitfall with disregarding commonly used ("easily" activated) circuits from suspicion is that doing so could cause some well-disguised MIs to be missed. As shown by Sturton et al., MI signals can be "piggybacked" on commonly-used circuits, and thereby evade detection, because they ride along on an "easily" activated circuit [2]. Based on this demonstration, we conclude that assuming MIs reside in *only* unused or rarely-used circuits does not support a complete, sufficient method of MI detection.

3. Summary

To conclude, current hardware security analysis efforts generally take one of two approaches:

- Assume that one design or processor sample is trusted, and compare other designs or physical samples to this reference.
- Look for elements of a design that are not frequently triggered, and single them out for further analysis.

In general, the approaches described in Section B do not clearly describe what constitutes *secure* or *insecure* behavior in a hardware design. We believe that it would be useful to adopt a more constructive method, whereby the design of a processor can be tested against some understandable, well-defined, stated criteria, such as a set of behavioral requirements enumerated in an overall security policy, as opposed to just testing for equivalence or identifying infrequently used circuits. In the following chapters, we develop a method aligned with this philosophy.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. ASSERTIONS AND THE PROPERTY SPECIFICATION LANGUAGE

> "How can one check a [program] 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* [emphasis added] which can be checked individually, and from which the correctness of the whole [program] easily follows."
>
> –Alan Turing, 1949 (Jones, *Annals of the History of Computing*, 2003)

A. INTRODUCTION, MAIN IDEAS, AND OBSERVATIONS

In Chapters VI and VII, we outline a novel method for detecting processor malicious inclusions. The method is based on the following general observations:

- A security policy specifies permitted and prohibited behaviors.
- Malicious inclusions, in the examples seen to date, often violate some behavioral restriction that is either stated or implied in a processor's architectural specification.
- If we can identify these behavioral restrictions in text statements, it should be possible to express them formally, so we can evaluate a particular processor's design against them.
- Formally defining behavioral restrictions can express a security policy, and may help us to detect malicious inclusions that violate it. In other words, the conjunction of all of the behavioral restrictions forms a security policy.
- The behavior of hardware systems can be expressed using assertions, for example using the Property Specification Language (PSL).
- Assertions are already used for functional verification, but can also be used to describe behaviors that a processor's designers feel should be prohibited or permitted.
- The conversion of PSL assertions into equivalent synthesizable "checkers" allows us to monitor hardware behavior, *using other hardware modules*, at runtime. This ability spans simulation, FPGA emulation, and fabrication.

Based on these observations, we next examine hardware security, in the context of assertions.

B. SECURITY AND ASSERTIONS

The word "secure," as applied to a computing system, may assume various meanings for different people. It may be considered synonymous with "trustworthy," for example, or defined in other terms. Some possibilities include:

- Made by people we trust, in places we trust, using methods we trust, as in the DARPA TIC program [53].
- Evaluated successfully against some standard, such as the Common Criteria [54].
- Functionally equivalent to some trusted reference sample or design [50].
- Containing information protection devices, such as encryption units.
- Containing detection systems and defenses, like anti-virus and updated software patches, to identify and protect against known attacks.

Though useful, we argue that such definitions are not sufficient for fully defining security. The security of a computing system—whether hardware, software, or both—should be measured against a well-defined set of security policies or properties. In essence, we propose a *mapping* from a set of stated security requirements in an architectural specification to an equivalent enforcement mechanism in the hardware design itself.

But how should one perform this *mapping*, to express security requirements at the hardware level? Since security requirements, in general, describe permitted and prohibited behaviors, it makes sense in hardware to use a construct like the *assertion*, because hardware assertions describe hardware behaviors. Assertions have been used by chip designers for many years to aid in the evaluation of functional correctness, and so-called *assertion-based verification* is common in the processor design industry today [44]. We believe it is natural to extend the use of assertions from functional verification to the definition and evaluation of security properties, and we propose a method for doing so.

Hardware assertions differ from software assertions in an important way, because hardware languages are primarily based on constructs that execute in parallel, whereas most software languages describe execution that is fundamentally sequential. In general, a software assertion is checked when the program execution arrives at the assertion point, but a hardware assertion is *continuously evaluated*, in parallel with the rest of the system's execution. As described by Eisner and Fisman, "Unlike assertions in other languages, [hardware] assertions are not embedded in the code or part of the code. Rather they are *about* the code, typically standing alone" [55].

Applying assertions in the context of hardware security is facilitated by two important developments:

- The standardization of PSL as a single language, rich in expressive power, specifically designed for formulating assertions for all the popular hardware-design languages [56].
- The evolution of techniques for efficiently synthesizing PSL-based *hardware assertion checkers*, hardware design units that can dynamically verify whether an asserted PSL property holds on the current execution [6].

The rest of Chapter VI contains a brief description of PSL. The construction of hardware assertion checkers for PSL is covered in Chapter VII.

C. PSL BACKGROUND AND DISCUSSION

Before the advent of *hardware* verification, assertions had long been used to verify the correctness of *software* programs, and the idea has been around for even longer. The assertion concept was introduced by Goldstine and von Neumann in 1947, according to Jones [57]. Instead of software, though, we wish to use assertions to describe and verify the behavior of hardware over time.

Above the physical level, processors and other hardware units are logical devices. They electrically represent ones and zeros on circuits, whose values are combined using gates performing functions like AND, OR, and NOT. So how does one describe the behavior of a logical system over time? Using temporal logic. Pnueli introduced the idea of describing the behavior of logical systems over time, in order to prove certain properties about them [58]. There are two basic categories of temporal logic: those that are linear (i.e., non-branching), like Linear Temporal Logic (LTL) [58] and those that permit branches (multiple possible futures) in time, like Computation Tree Logic (CTL) [59]. PSL, which incorporates both linear and branching logics, was developed with verification of hardware specifically in mind, though it can be used in software verification as well [60].

PSL evolved from a language called Sugar, developed by IBM [55]. Sugar, used for model-checking, was so-named because it featured a great deal of "syntactic sugar," so that temporal logic formulas could be written in a way that is more easily understood. The initial standardization effort was led by Accellera, and PSL was standardized by IEEE in 2005; the most recent version was approved in 2010 [56]. The purpose of PSL is described in its specification (emphasis added):

> *PSL is a language for the formal specification of hardware*. It is used to describe properties that are required to hold in the design under verification. PSL provides a means to write specifications that are both easy to read and mathematically precise. It is intended to be used for functional specification on the one hand and as input to functional verification tools on the other. Thus, *a PSL specification is an executable specification of a hardware design.*

Though PSL supports both linear-time logic and branching-time logic, our focus is on the former, given our interest in verifying properties dynamically, at runtime. The difference is best summarized by Eisner and Fisman [60]:

> In branching-time logics such as CTL. . . time is *branching*. That is, the semantics are given with respect to the state of the model, and every possible future of that state is considered. In linear-time logics such as LTL, time is *linear*. That is, the semantics are given with respect to a set of ordered states (a path) in the model, and every state has a single successor. In theory, this is a very big deal. The complexity of branching time model checking is better (lower) than that of linear time model checking, the expressive power of the two is incomparable, and of course, only linear time makes sense for dynamic and runtime verification. In practice, though, the issue is not such an important one. The overlap between linear and branching time is a large one, and the vast majority of properties used in practice belong to the overlap. Furthermore, there is a simple syntactic test that can be used to confirm that a syntactically similar CTL/LTL formula pair is equivalent. . . for instance, [formulas in the] *Simple Subset* of PSL obey this test.

For the reasons cited, we focus on the portion of PSL that derives from LTL; it is called the Foundation Language (FL). We do not use the other portions of PSL, called the Optional Branching Extensions (OBE), which derive from CTL.

D. PROCESSOR PHYSICAL INTERPRETATION

Before discussing the elements of PSL, we describe how real, physical circuits in a processor come to be modeled by the boolean primitives in PSL. At the physical level, a processor is analog, rather than digital, with continuous values for voltage, current, resistance, capacitance, etc. However, a processor may be described as a digital entity, representable by the model above, by way of a number of abstractions.[8]

First, we consider only the voltage of the signals that comprise the processor as our discrete system state. There will normally be an asserted voltage and a de-asserted voltage for each signal; also, circuits may be designated asserted-high (higher voltage indicates logical "1," or *true*), or asserted-low (lower voltage indicates logical "1," or *true*) in a design. In addition, there may be intermediate voltage representations. For example, the Verilog hardware design language models signals using a four-valued logic; the interpretations we use in PSL input sequences are listed in Table 4.

Symbol	**Physical Interpretation**	**Logical Interpretation in the Model**
1	true	true
X	unknown	false
Z	high impedance	false
0	false	false

Table 4: Verilog signal voltage interpretations.

Verilog permits the use of four-valued logic and defines its behavior in logic gates [61], but when observing signal values for use in PSL assertions, we need to map them to only two values (*true, false*). To do so, we adopt the convention that an input signal of "1" represents *true* and any other value, including "0," "X," and "Z," represents *false*.

Next, we assume that there is a clock driving the processor circuits, and that all the signals being observed either change value or remain steady at the occurrence of a clock cycle. We assume that there exists some stable period that is long enough to sample the values of the signals. PSL permits the use of multiple clock domains and also has unclocked semantics, but for simplicity we assume a single clock domain [62].

Under these assumptions, a processor can be represented by a set of synchronous two-valued inputs, and therefore modeled by PSL's boolean values. See Figure 9 for an example. In the figure, at top, signals *a* and *b* undergo periods where their physical value is undefined, high impedance, metastable, or in transition from one state to another; at bottom, their clean logical interpretation is shown. There is a single clock domain, with transitions on the rising edge.

[8] We assume the processor is not of the analog-mixed-signal (AMS) variety.

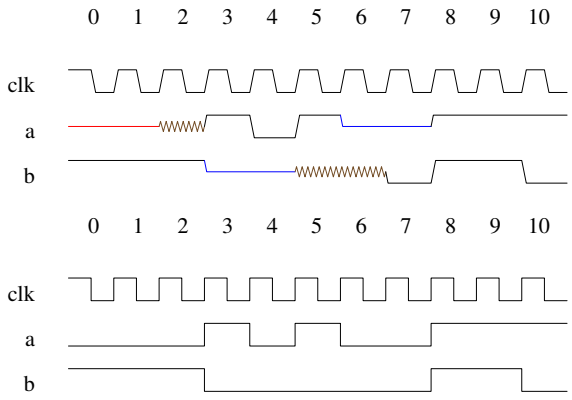


Figure 9: Example of a processor's physical circuit values (top) and their logical interpretation in the model (bottom).

E. ELEMENTS OF PSL

Because we would like to use PSL to describe the behavior of processors over time, a brief description is in order. PSL is broken up semantically into four layers, the *boolean layer*, the *temporal layer*, the *verification layer*, and the *modeling layer* [56].

The *boolean layer* is composed of boolean expressions, which can evaluate to *true* or *false*. If a processor signal is "asserted-high," the boolean interpretation of the circuit is true when asserted and false when not asserted. For example, "`!b && c`" is a boolean-layer expression for "not b and c."

The *temporal layer* provides ways to describe the behavior of boolean expressions over time. For example, "`always c`" says that the boolean evaluation of "`c`" should always be true (in every clock cycle), while "`next a`" says that "`a`" should evaluate to true in the next clock cycle.

The *verification layer* is used to describe what a verification tool should do with temporal properties. For example, it can *assert* them, meaning they are required to hold, or it can *assume* that some properties hold, in order to provide an invariant to check whether other properties will hold as well. The verification layer also allows the various directives to be assembled into verification units, or *vunits*, which can be attached to associated hardware design units.

The *modeling* layer provides a way of modeling the behavior of design inputs, and also allows the of declaration of local variables and auxiliary signals.

Our focus will be on the boolean and temporal layers, where sequences, properties, and assertions are primarily defined.

1. Basic Temporal Operators

The Foundation Language is composed of two parts: LTL style, which inherits from LTL, and SERE style. SERE stands for Sequential Extended Regular Expression [56]. We discuss LTL-style operators first, mirroring the informal descriptions of Eisner and Fisman [55].

Note that PSL makes a distinction between *sequences*, *properties*, and *assertions*. A *sequence* is merely an ordering of events. A *property*, on the other hand, represents some type of *temporal obligation*, and an *assertion* dictates that the obligation should be met. For example, "`a`" represents a *sequence* defined by one occurrence of the boolean expression "`a`." The property "`always a`" indicates that "`a`" must be true on every clock cycle if the property is to hold, and the assertion "`assert always a`" applies a directive to tell the verification unit that the property must hold.

Two of the most common operators in PSL are "`always`" and "`never`." Most PSL properties use one or the other. If we assert that a boolean condition must hold without saying "`always`" or "`never`," then by the PSL semantics the condition is only required to hold on the first clock cycle. For example, "`assert a`" holds if signal "`a`" is high on the *first* clock cycle, even if signal `a` is de-asserted thereafter. However, "`assert always a`" indicates that signal "`a`" must be high on *every* clock cycle, and "`assert never a`" indicates the opposite.

Another common temporal operator is "`next`." It indicates that a property will hold if its operand holds at the next clock cycle. There are many variations on the "`next`" operator. For example, "`assert always b → next[2:3](c)`" says that whenever signal "`b`" evaluates to true in a given clock cycle, signal "`c`" must be true from the second through the third clock cycles afterward.

PSL also features operators for "`until`" and "`before`." As its name implies, the "`until`" operator requires that the left operand remain true until the right operand becomes true, as in the example in Figure 10 [55].

The assertion `assert always (req → next (busy until done))` holds on this trace.

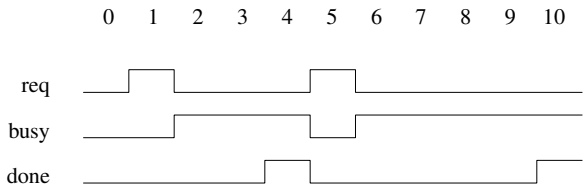


Figure 10: Example use of the "`until`" operator.

To satisfy the assertion, each time the request signal "`req`" goes high, starting in the next cycle, "`busy`" must be true until at least when "`done`" is true. Similarly, the "`before`" operator specifies that the left operand must be true strictly prior to the right operand being true.

PSL also has an operator "`eventually!`," which specifies that its operand must be true at some time in the future. It uses the exclamation point (!) to indicate that it is a *strong* operator; strong and weak forms are discussed next, in Section 2.

2. Strong and Weak Operators

PSL uses strong operators to indicate that a pending requirement, or "termination condition," must eventually be satisfied, and that completing the trace without satisfying the condition results in an assertion failure. For example, the "`next`" operator, in its *weak* form, requires that its operand be true the cycle after the left-hand operand is true. If the trace ends before the specified *next* cycle can occur, no failure is reported. However, the *strong* form ("`next!`") requires that the trace include the specified *next* cycle; if the sequence terminates early, the obligation has not been met, and an assertion failure results.

An illustration is given in Figure 11. In the first assertion, every instance of "`a`" being true requires that, in the second clock cycle following, "`b`" is true; because the weak form "`next`" is used, the termination of the trace at cycle 10 does not result in an assertion failure, since the next "`b`" wasn't required until cycle 11. In the second assertion, the strong form ("`next!`") is used; the obligation created by signal "`a`" at cycle 9 has not yet been fulfilled when the trace terminates, so the assertion fails.

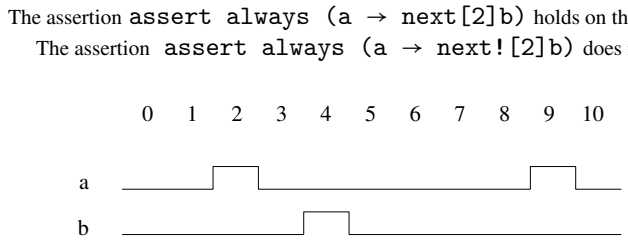


Figure 11: Strong and weak forms of the "`next`" operator.

There are also strong forms of the PSL operators "`before`" ("`before!`") and "`until`" ("`until!`"). The "`eventually!`" operator has only a strong form. The "`always`" and "`never`" operators have only weak forms, since they have no implied "terminating" conditions to relieve their obligations [56].

3. Operator Comparison

PSL's inheritance from LTL is visible in the similarity of its temporal operators. Table 5 lists the original LTL operators, with their PSL equivalents [6].

LTL Operator	PSL Equivalent
X	next
X!	next!
G	always
F	eventually
U	until!
W	until

Table 5: LTL operators, and their PSL equivalents.

4. SEREs

In addition to the basic temporal operators, PSL has "SERE-style" temporal operators, so named because of their resemblance to traditional regular expressions. The term SERE is an acronym for "sequence-extended regular expression," since SEREs derive some of their characteristics from *regular expressions* and some from the notion of temporal *sequences*.

The degenerate SERE "`[*0]`" does not accept any sequence. Boolean expressions are the building blocks of SEREs, and a boolean expression by itself is a SERE (for example, "`{b}`" is a SERE representing one occurrence, or clock cycle, in which the boolean expression "`b`" evaluates to true).

a. Concatenation and Fusion

Like traditional regular expressions, SEREs can be joined together by concatenation. SEREs use the semi-colon (;) to indicate concatenation over consecutive clock cycles. The SERE "`{a;b;c}`" indicates that "`a`" is true in one clock cycle, "`b`" is true in the next, and "`c`" is true the cycle after "`b`."

Similarly, PSL provides for the fusion of two sequences, in which the final clock cycle of the first sequence is simultaneous with the first clock cycle of the second sequence. The fusion operator is the colon (:). For example, the SERE "`{a;b}:{c;d}`" is semantically equivalent to the SERE "`{a;b∧c;d}`".

b. Suffix Implication

The suffix implication operators $\mapsto$ and $\models\!\!\Rightarrow$ provide a means for connecting two SEREs in the form of a temporal obligation. When the left operand sequence holds, it triggers a requirement that the right operand sequence must subsequently hold. In the case of non-overlapping suffix implication ($\models\!\!\Rightarrow$), the righthand sequence begins the clock cycle *after* the lefthand sequence completes. In overlapping suffix implication ($\mapsto$), the final clock cycle of the lefthand sequence must *overlap with* the first clock cycle of the righthand sequence. The assertion "`assert always a` $\models\!\!\Rightarrow$ `b`" requires that, any time "`a`" is true, "`b`" is obligated to be true the following clock cycle, or the assertion fails.

In addition to the two suffix implication operators ($\mapsto$, $\models\!\!\Rightarrow$), PSL supports traditional logical implication ($\rightarrow$) at the boolean layer. The difference between the two types of implication is in their operands and their timing semantics. For logical implication, both operands are booleans, and the implication is evaluated only in the *current* clock cycle (as usual, a$\rightarrow$b is syntactic sugar for $\neg$a$\vee$b). In suffix implication, the operands are sequences, and evaluation takes place over multiple clock cycles. In non-overlapping suffix implication ($\models\!\!\Rightarrow$), when the left-hand sequence has been observed, completing in clock cycle *n*, it triggers an obligation that the right-hand sequence be observed, starting in clock cycle *n+1*, else the implication will not hold. Overlapping suffix implication ($\mapsto$) is similar, but the end of the left-hand sequence and the beginning of the right-hand sequence overlap in cycle *n*.

Note that the right arrow ($\rightarrow$) has multiple meanings that depend upon context. It is also used with PSL repetition operators, explained in the next section. In the context of repetition, the right arrow is called the "goto" operator, has only a right-hand operand, and always appears inside square brackets next to a numeric count or range, as in "`(b)[→5]`."

c. Repetition

Within a SERE, boolean expressions can be repeated in a number of ways. The repetition operator "`r[*n]`" indicates that SERE "r" is true for *n* consecutive clock cycles. The repetition operator can also specify a range; the expression "`r[*m:n]`" indicates that SERE "r" is true from the *m*th through the *n*th consecutive clock cycles. The "`[*]`" operator represents the Kleene closure of a SERE (zero or more instances), while the "`[+]`" operator indicates non-empty closure (one or more instances) of a SERE "`r`," in the format "`r[*]`" or "`r[+]`," respectively.

Boolean expressions can be repeated using the "count repetitions" and "goto repetition" operators. The count-repetitions operator "`b[=n]`" counts the number of clock cycles in which "`b`" is true, but does not require the repeated occurrences of "`b`" to necessarily be in consecutive clock cycles. The "goto repetition" operator, "`b[→n]`," is slightly different from count-repetitions, in that it only holds exactly when the *n*th repetition occurs. The difference is shown in Figure 12. The assertion "`assert {a} ⊨ {b;c[=2];d}`" holds on this trace, but the assertion "`assert {a} ⊨ {b;c[→2];d}`" does not. That is because the current evaluation of "`c[→2]`" holds only in cycle 5, whereas the current evaluation of "`c[=2]`" holds from cycle 5 until an instance of "`d`" is observed, or the end of execution.

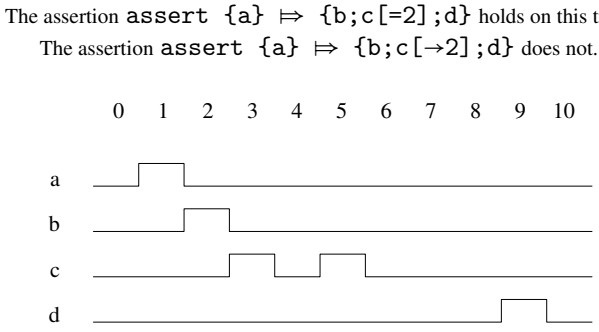


Figure 12: The difference between count-repetitions (`b[=n]`) and goto-repetition (`b[→n]`).

d. Conjunction and Disjunction

SEREs may also be combined using conjunction and disjunction. The disjunction operator is "`|`." A compound SERE "`{a}|{b}`" holds whenever "`{a}`" holds or "`{b}`" holds.

The SERE conjunction operators are "`&`" and "`&&`." A conjuncted compound SERE holds if, starting at the same clock cycle, both its left and right operand SEREs hold. In addition, when using the *length-matching* conjunction operator "`&&`," both operand SEREs must hold over an *equal* number of clock cycles for the conjunction to hold. When using the *non-length-matching* conjunction operator "`&`," the operand SEREs may hold over different sequence lengths; the combined SERE holds during the clock cycle when the longer of the two operand SEREs holds.

5. Safety and Liveness Properties

In Chapter IV, we described traditional high-level security policies, and noted their primary constructs (e.g., subject, object, security label) generally exist at the *software* level of abstraction in computing systems. What security policies or properties then might we describe at the *hardware* level? As suggested by Schneider [34], we can use *safety* and *liveness* properties. Liveness and safety are well-recognized property types in security theory [63]. Because they do not necessarily rely on higher-level abstract constructs, safety and liveness properties *are* simple enough to be applicable at the hardware level, so it is no surprise the PSL specification contains definitions for both of these property types, in context [56]:

- "A *safety* property is a property that specifies an invariant over the states in a design. The invariant is not necessarily limited to a single cycle, but it is bounded in time. Loosely speaking, a safety property claims that 'something bad' does not happen. More formally, a safety property is a property for which any path violating the property has a finite prefix such that every extension of the prefix violates the property."
- "A *liveness* property is a property that specifies an eventuality that is unbounded in time. Loosely speaking, a liveness property claims that 'something good' eventually happens. More formally, a liveness property is a property for which any finite path can be extended to a path satisfying the property."

Because of theoretical restrictions on what is enforceable at runtime [34], our primary focus in constructing properties will be on *safety* properties. Fortunately, they are easy to recognize in PSL: "[A property] that contains only non-negated weak operators is a safety property" [56].

6. The Simple Subset

PSL is a rich language, capable of expressing properties that are very difficult to evaluate dynamically [55]. In order to be able to describe properties that do facilitate runtime enforcement, the language's designers created the Simple Subset. The Simple Subset embraces the notion of monotonically advancing time through the evaluation of a formula, from left to right. In other words, time "flows from left to right" through the formula. Informally, if there is an entity x within a PSL formula whose value we desire to know at clock cycle n, then we need not know anything about the values of entities to the *right* of x in the formula in order to evaluate it; it is sufficient to know the values of the entities to the *left* of x in the formula at clock cycle n. In order for a PSL property to be in the Simple Subset, it must obey the restrictions listed in Table 6 [56]. Operators not listed may be used without restriction.

According to Eisner and Fisman, focusing our attention on the Simple Subset of PSL is not exceptionally limiting. They note, "Many properties not in the Simple Subset can be rewritten into the Simple Subset," and "We have never come across a hardware design in which a needed property could not be expressed within the bounds of the Simple Subset" [55]. PSL support by commercial tools is also generally limited to the Simple Subset, or a further subset of it [55]. The primary research to date on creating synthesizable PSL assertion checkers, by Boulé and Zilic, employs only the Simple Subset [6], and we follow that example, as discussed in Section F. Portions of PSL not in the Simple Subset are not as useful for dynamic verification, but are useful in static verification.

PSL Operator	Simple Subset Restriction
`!`	Operand must be boolean
`never`	Operand must be boolean or sequence
`eventually!`	Operand must be boolean or sequence
`||`	At most one operand may be non-boolean
→	Lefthand side must be boolean
↔	Both operands must be boolean
`until, until!`	Righthand side must be boolean
`until_, until!_`	Both operands must be boolean
`before*`	Both operands must be boolean
`next_e`	Operand must be boolean
`next_event_e`	Righthand operand must be boolean

Table 6: PSL Simple Subset restrictions.

F. SYNTHESIZABLE PSL ASSERTION CHECKERS

Before the advent of synthesizable assertion-checkers, which permit the assertions to be expressed and evaluated in physical form, engineers could only use assertions to check the behavior of hardware designs in software simulation. However, a principal limitation of using assertions is the "simulation bottleneck" [6]. Because a software simulation operates *linearly*, modeling a hardware design that has many components that work in *parallel*, simulation of large hardware designs becomes prohibitively time-consuming. This is especially true when using assertions, since software-based assertion checkers may require a lot of data-gathering and a large number of "thread" instantiations to check instances of an assertion over the course of a simulation run [6]. To improve performance, it is useful to bypass the simulation bottleneck, and perform verification in hardware, such as FPGA emulation, where possible, using synthesizable assertion checkers.

Modern high-level hardware design is principally done using a Hardware Design Language (HDL). HDLs describe a hardware module's interface with other modules (its inputs and outputs), and how the module behaves, in terms of the internal structure and interaction of the signals. Popular HDLs include VHDL and Verilog, which derive their look and feel from the software languages Ada and C, respectively [61], [64]. What's important to note is that *not all constructs in an HDL are synthesizable* [61], [64]. Synthesis is the process of translating the high-level HDL specification into a low-level form, such as a netlist. The *netlist* is a combined low-level physical representation (signals, or "nets") of all the hardware modules as they execute the behavior described by the HDL. Once synthesized, a design is ready for some type of physical mapping, as in the place-and-route routine of an FPGA or the floorplanning process for silicon designs. In an effort to create hardware assertion checkers, we need to use only the portions of languages like VHDL or Verilog that are synthesizable; each language also has non-synthesizable language constructs that are useful in simulation, but are not supported for translation into physical hardware elements by commercial tools. For example, the VHDL command "`wait for 5 ns`" is valid in simulation, but is not in the VHDL synthesizable subset [64].

IBM's Formal-Checkers (FoCs) tool was the first to attempt synthesis of hardware assertion-checkers [60]. FoCs was based on Sugar, a PSL precursor. Since then, and with the advent of the PSL specification, most of the research in generating resource-efficient hardware assertion-checkers has been done by Boulé and Zilic [6], who refer to their tool by the name MBAC. Findenig also produced a tool, called SynPSL, for

a smaller subset of PSL assertions, based on the methods outlined by Boulé and Zilic, and demonstrated its use [4]. Collectively, we refer to software tools like FoCs, MBAC, and SynPSL, which generate synthesizable hardware modules that in turn check assertions at runtime, as "checker generators." Our own checker generator is discussed at the end of Chapter VII.

VII. GENERATING PSL-BASED ASSERTION CHECKERS

> "If a program has not been specified, it cannot be incorrect; it can only be surprising."
>
> –Young, Boebert, and Kain, "Proving a Computer System Secure," 1987.

A. INTRODUCTION

Since PSL can be used to describe the behavior of hardware over time, and since hardware "checkers" can be synthesized for many PSL assertions, it seems natural to see if we can use PSL to describe, and implement checkers for, the behavioral restrictions on a processor's behavior. Bilzor, Huffmire, Irvine, and Levin gave the following examples of some processor behavioral requirements that a designer might choose to specify [65]:

- Only a process running in supervisor mode may modify the control/special registers.
- Execution transfers from user mode to supervisor mode must only occur through specifically defined processor gates, interrupt calls, or exceptions.
- A memory segment labeled with a certain privilege level may not be modified or read by a process labeled with a lower privilege level.

Where would we expect behavioral requirements like these to be enumerated? We propose that, in order for a hardware design to be more meaningfully called "secure," the security requirements should be clearly stated somewhere, so that an implementation can be compared against them. In the case of a general-purpose processor, that place should be the *architectural specification*.

1. Architecture and Implementation

In modern processors, the *architecture* defines a general set of structures and behaviors—the instruction set, the registers, addressable space, etc. A processor engineer may choose to *implement* the architecture in any of a number of different ways, as long as it conforms to the architectural specification. The difference is described in the Architectural Manual for the MIPS processor architecture [66]:

> *Architecture* refers to the instruction set, registers and other state, the exception model, memory management, virtual and physical address layout, and other features that all hardware executes. *Implementation* refers to the way in which specific processors apply the architecture.

It is natural to focus on the possibility a malicious adversary can produce a malicious inclusion (MI), but it is also possible that a manufacturer intentionally leaves some unpublished backdoor in a processor, e.g., for easier post-silicon debugging [16]. It is also possible that a design error creates an unintentional vulnerability. What is the difference between these cases, and what makes a behavior "malicious"? For illustration, consider the following questions:

- A processor may have significant access vulnerabilities, bypass mechanisms, and powerful debugging features, etc., but if they are all intentionally part of the design, and not prohibited by the architectural specification, are they necessarily "malicious"?
- If a circuit fails an unused circuit identification (UCI) test [2], is the circuit malicious, part of an inefficient or erroneous design, or is it just highlighted as a result of an incomplete testbench?
- If the architecture purposely provides an instruction to enable or disable memory segment checks from user mode, is that an unwise feature of the architecture, or a malicious inclusion?
- If an implemented processor circuit does something that is neither explicitly allowed nor prohibited by the architectural specification, can that circuit be considered "malicious," or is the architectural specification just incomplete?

In Chapter III, we define an MI as "an unauthorized, undocumented modification to a piece of hardware, or hardware design unit, that circumvents or subverts some portion of the hardware's functionality." One of the main ideas of this research, outlined at the beginning of Chapter VI, is that processor MIs may be detectable *as violations of behavioral restrictions in the architectural specification and other governing documents*.

A conceptual view of an architectural specification, with and without explicitly stated behavioral restrictions, is given in Figure 13. The architecture in (a) has no behavioral requirements for us to *map*, but the one in (b) does. One could argue that any implementation of the architecture in (a) will be *vacuously* secure, because the architecture imposes no behavioral requirements, and we would consider the specification to be incomplete. On the other hand, any implementation of (b) will need to adhere to the stated behavioral requirements, in order to be called secure.

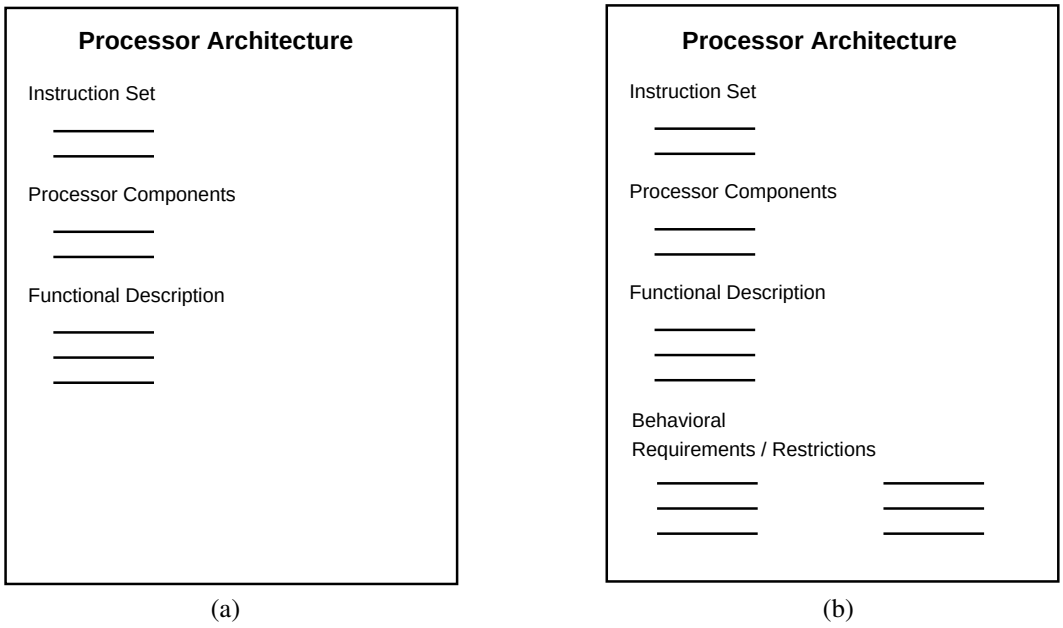


Figure 13: Conceptual view of an architectural specification, without (a) and with (b) explicit behavioral requirements.

In its broadest sense, a security policy is a predicate on executions [34]. Together, a set of behavioral requirements, e.g., a set of prohibited behaviors, forms a security policy. If a processor's execution exhibits any of the prohibited behaviors, the security policy is violated by that execution.

What kind of elements in a processor will usually need behavioral restrictions imposed on them? Based on the description of malicious inclusions in Chapter III, some hardware modules suggest themselves for consideration:

- Circuits controlling *access*, such as access to memory segments or I/O devices
- Circuits related to *privilege level* or *operating mode*, where one privilege level or mode has greater power (or fewer restrictions) than another
- Circuits that support *special supervisory functions*, like virtualization
- Circuits that can circumvent or modify normal processor operations, such as debug circuits, performance and power regulation circuits, etc.
- Circuits that control *input* and *output*

Though not exhaustive, this list is representative of many of the circuits subverted in malicious inclusions demonstrated to date, as presented in Chapter III. We would expect that these kinds of circuits would be strong candidates for behavioral restrictions in a processor architecture.

2. Prohibited Behaviors

A processor's execution can, in general, be described as a sequence of states. At any given clock cycle, the state changes, as the digital signals of the processor transition to new sets of "1" or "0" (*true* or *false*) values. From the beginning of execution, usually at a reset, the processor moves through a sequence of states. Each sequence may be described as permitted or prohibited. One important aspect of an Execution Monitor (EM), which evaluates the runtime behavior of a system against a security requirement, is that any given sequence must be either permitted or prohibited; it cannot be neither, or both. Schneider describes a security policy as a predicate P on sequences [34]; for any one sequence, P must either be *true* or *false*.

Because runtime enforcement mechanisms in an EM require sequences to be permitted or prohibited, we could try to expressly describe every possible sequence that a processor could execute, but of course that is impractical. Instead, we can take one of two simpler approaches:

- Whitelist Approach: Describe all permitted behaviors (sets of sequences) explicitly, and stipulate that everything else is prohibited.
- Blacklist Approach: Describe all prohibited behaviors explicitly, and stipulate that everything else is permitted.

Given the large number of possible permitted behaviors in a processor, we expect it will usually be simpler to take the latter approach (blacklist). Several possible scenarios are illustrated in Figure 14. In cases (a) and (b), all possible behaviors are either permitted or prohibited; neither case is interesting nor requires a security

policy [34]. Case (c) shows the whitelist approach, in which permitted behaviors are explicit and all else is prohibited. Case (d) shows the blacklist approach, in which prohibited behaviors are explicit and all others are permitted.

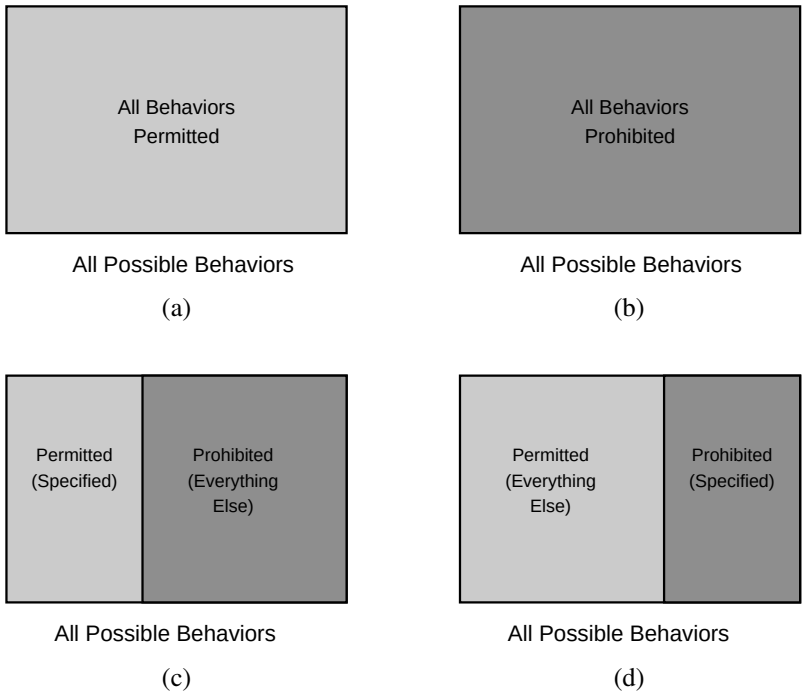


Figure 14: Permitted and prohibited behaviors.

Superficially, there is a resemblance between the behavioral blacklist approach described here and similar tactics used in software, specifically for malware detection. Software anti-virus programs look for behavioral signatures, for example, and use a blacklist approach against them [67]. Network behavior can also be governed in this manner. However, there is an important difference when applying this approach in hardware. In hardware, the set of signals (whose behavior is being examined) does not change over time, because a processor, once fabricated, is in a fixed configuration (aside from any reprogrammable firmware). Software, on the other hand, assumes ever-evolving configurations, by way of operating system updates, third-party software, add-ons, etc. This significant additional complexity makes signature-based blacklist approaches to software security extremely difficult [68], because the set of blacklisted behaviors must continually evolve. Because a processor's circuit configuration is fixed, there is no need to constantly update the list of prohibited behaviors, as in software, and the problem is therefore more manageable. In general, we will adopt the blacklist philosophy in hardware, seeking to identify examples of explicitly prohibited behaviors, and permitting all others, since it is impractical to explicitly describe every possible permitted behavior in a real system.

3. Requirements and Verification

When evaluating a system, the basis against which it is evaluated is some set of *requirements*. These will take on a form that can be described as *positive* or *negative*. For example, "Property X must always hold" is a positive requirement, and "Property Y must never be observed" is a negative requirement, similar to the whitelist and blacklist discussion in the previous section. It may be possible to express some requirements equally well in either positive or negative form.

The process of evaluating a system against a set of requirements is called *verification*, which can be defined simply, in accordance with similar definitions [44], [69], [70] in the literature:

- Verification: the process of demonstrating that the *requirements* of a system are met by a particular *implementation*.

In assertion-based verification, we express each *requirement* in the form of one or more *assertions*. The process of verification, then, involves evaluating whether an assertion does, or does not, hold for a particular implementation of a design. As discussed in Chapter V the verification can be *static*, as in formal methods or model checking, or *dynamic*, as in simulation or FPGA emulation. As outlined in the rest of this chapter, our research focuses on the dynamic evaluation of assertions, with each assertion representing a requirement that some *prohibited* behavior not be observed in the hardware design. As such, according to the definition, our method is a form of verification.

B. CONVERTING TEXT TO PSL ASSERTIONS

Before our PSL assertions can be converted into equivalent synthesizable checkers, the assertions must be written. Writing precise assertions is a difficult task, the most tedious and challenging part of this method. The idea is to translate textual English-language behavioral requirements, which may be expressed in a high-level or abstract manner, from the architectural manual into PSL assertions that are appropriate for a given hardware implementation. If the requirements are stated at a high level of abstraction, it may be necessary to first translate them to a low-level intermediate textual format, in some cases, before creating the PSL assertion.

Boulé and Zilic present an example of this type of text-to-PSL conversion [6], which we adapt below. The example involves requests for access to some resource.

- High-level requirement: "Requests must be granted within five cycles, barring a reset."
- Low-level requirement: "When the *request* signal goes from low to high, 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."
- PSL assertion: "`assert always ({!req;req} ⇻ {req[*0:5];gnt}) abort rst`"

The PSL representation is more succinct than the low-level text description. Also, PSL's formal semantics are unambiguous, meaning that any ambiguities inherent to the English-language expression must be resolved during the conversion [56]. For example, the phrase "unless a reset occurs" is ambiguous in its reference; it

could refer to the obligation that the grant signal occur within 5 clock cycles, it could refer to the obligation that the request signal remain high in the interim, or it could refer to both. Translating the requirement from text to PSL necessarily removes the ambiguity, but doing so may require a good deal of knowledge and inference on the part of the translator.

When a behavioral requirement is translated from a text description into a PSL formula, the formula will have to obey the Simple Subset restrictions, due to the limitations of the established checker-generator method (PSL checker-generators not limited to the Simple Subset would be valuable future work).

In some cases, a high-level property may be constructed from two or more low-level properties. This is possible using PSL's property conjunction and disjunction operators, which are similar to the sequence conjunction and disjunction operators discussed earlier. In other words, *property* `||` *property* results in another property (either may hold), and so does *property* `&&` *property* (both must hold); using these combinations, more complex, higher-level properties can be constructed.

When mapping text to formulas, a complication sometimes arises, due to the hierarchical nature of hardware designs. When mapping a behavioral requirement to the named signals in a design which carry out the behavior in question, there are often multiple functionally equivalent instances of some signals in the design hierarchy. For example, consider the *clock* and *reset* signals, present in almost all processor designs. These are normally distributed from higher-level design units to lower-level design units (e.g., a clock tree for a clock domain). This hierarchical distribution of functionally equivalent signals is also common for *enable* circuits, like *read-enable* or *write-enable*. If there are multiple equivalent instances of a signal in a design hierarchy, and that signal implements some function that is described in a textual security requirement in the architectural specification, then PSL assertions will often have to be created for *each* hardware module carrying an instance of that signal.

How do we know if all the requirements that *ought to* be in the architectural specification are there? This is more of a philosophical question, and will depend on the intent of the architect, as well as the purpose of the hardware. It is an important question to answer, but is not in the scope of this investigation.

How do we know when all the stated textual requirements of the architecture have been expressed in terms of PSL formulas? Answering this question efficiently is an open research area; our current method requires going through the entire architectural specification section by section, and performing text-to-PSL translations like the one in the example above, when a behavioral requirement is encountered. It would be useful future work to add some automation to this part of the method, which can otherwise be tedious. Boulé and Zilic comment, "One question that often arises with new [verification] practitioners is: *How many assertions do I need to write*? The answer is not an easy one" [6]. Adding to the difficulty, some textual requirements may only be implied, or stated incorrectly, ambiguously, or incompletely. *A clear and complete statement of any behavioral restrictions in the architecture is necessary for successful application of our method*. Where the restrictions are stated incompletely or only implied, the specification writer may need to apply expertise and inference to construct the low-level, detailed requirements and translate them into assertions.

C. CONVERTING PSL ASSERTIONS INTO SYNTHESIZABLE CHECKERS

Once the text-based behavioral requirements are stated formally in PSL formulas, we can begin the process of converting the formulas into equivalent, synthesizable checkers. Throughout this section, we refer to the research of Boulé and Zilic, who have written extensively on converting PSL assertions into synthesizable form. With a few minor exceptions, we follow their basic method, outlined in the following steps:

- Parse the PSL formula.
- Given a parsed PSL formula, apply a set of rewrite rules. Often more than one rewrite rule may be applied to a formula. These rewrite rules convert PSL formulas from a wide variety of syntax into a small number of "base cases." This is possible because of the large amount of "syntactic sugar" in the language, which means that many PSL constructs are not in their most primitive form. The rewrite rules convert formulas into more-primitive forms.
- Consider the rewritten PSL formula in parse-tree format. Starting with the boolean-layer expressions at the leaves, create simple two-state automata that accept the boolean expressions. Working from the leaves of the parse tree to the root in depth-first-search order, combine the automata of the left and right children of each node into a single automaton for that node. Combine the automata using various rules for sequence concatenation, fusion, disjunction, conjunction, etc. Once the combination has terminated at the root of the tree, a single automaton representing the PSL expression remains.
- Convert the automaton into an equivalent hardware unit using an HDL, such as Verilog.

In the following section, we describe each phase of the conversion process in detail.

1. Rewrite Rules

First, we parse the PSL formula. The parser is described in Section D. After the parse tree for the formula is generated, we pass it through a system of rewrite rules.

There are two sets of rewrite rules, one for SEREs and one for properties. Many of the rewrite rules derive from the semantic definitions of the operators in question in the PSL specification. However, in some cases the rewrite rules used here differ from those given in the PSL specification because of the restrictions imposed by staying in the Simple Subset (i.e., a rewrite formula should not convert a Simple Subset PSL formula into a *non*-Simple Subset PSL formula). Some rewrite rules actually convert one form of an operand into a more complex form, rather than a simpler form; the purpose of this is to minimize the number of operand forms that need to be supported overall (minimize the number of base cases). Very often, performing one rewrite on a formula will result in the need to apply another rewrite rule; the rules are applied repeatedly until no more rules can be applied, and the formula is finally in a base-case form.

In the tables that follow, "`r`" is a SERE, "`b`" is a boolean, "`c`" is a positive integer (count), "`l`" and "`h`" are positive integers (range low to high, `l` $\leq$ `h`), and "`p`" is a property.

a. Property Rewrite Rules

The rewrite rules for properties are listed in Table 7 [71].

Original	Rewrite
b || P	(~b)→p
b→p	{b}↦p
always p	{[+]}↦p
never r	{[+]:r}↦false
next p	next[1]p
next! p	next![1]p
eventually! r	{[+]:r}!
p until b	{(~b)[+]}↦p
p until! b	(p until b)&&({b[→]}!)
b1 until_ b2	{(b1)[+]:b2}
b1 until!_ b2	{(b1)[+]:b2}!
b1 before b2	{(~b1&~b2)[*];(b1&~b2)}
b1 before! b2	{(~b1&~b2)[*];(b1&~b2)}!
b1 before_ b2	{(~b1&~b2)[*];b1}
b1 before!_ b2	{(~b1&~b2)[*];b1}!
next[c](p)	next_event(true)[c+1](p)
next![c](p)	next_event!(true)[c+1](p)
next_a[l:h](p)	next_event_a(true)[l+1:h+1](p)
next_a![l:h](p)	next_event_a!(true)[l+1:h+1](p)
next_e[l:h](p)	next_event_e(true)[l+1:h+1](p)
next_e![l:h](p)	next_event_e!(true)[l+1:h+1](p)
next_event(b)(p)	next_event(b)[1](p)
next_event!(b)(p)	next_event!(b)[1](p)
next_event(b)[c](p)	next_event_a(b)[c:c](p)
next_event!(b)[c](p)	next_event_a!(b)[c:c](p)
next_event_a(b)[l:h](p)	{b[→l:h]}↦(p)
next_event_a!(b)[l:h](p)	next_event_a(b)[l:h](p)&&{b[→h]}!
next_event_e(b1)[l:h](b2)	{b1[→l:h]:b2}
next_event_e!(b1)[l:h](b2)	{b1[→l:h]:b2}!
r↦p	{r;true}↦p

Table 7: Property rewrite rules.

After the property rewrite rules are repeatedly applied and no further simplification is possible, only ten base cases remain. During the rewrite process, many properties will be converted into a SERE-style format, then undergo SERE rewrite rules. The property base cases are shown in Table 8 [71].[9]

[9]Further simplification is possible, as discussed in the following sections.

b
r
p abort b
p1 && p2
b↔b
r↦p
(p)
r!
!b

Table 8: Property base cases.

Once in their base-case form, properties are fairly straightforward to implement in automata representations, as described later in this chapter.

b. SERE Rewrite Rules

The rewrite rules we use for SEREs are listed in Table 9 [72].

Original	**Rewrite**
r[+]	r;r[*]
r[*0]	[*0]
r[*c]	r;r;...;r *(c times)*
r[*l:h]	r[*l]|...|r[*h]
b[→]	{~b[*];b}
b[→c]	{b[→]}[*c]
b[→l:h]	{b[→]}[*l:h]
b[=c]	{b[→c]};(~b)[*]
b[=l:h]	{b[→l:h]};(~b)[*]
r1 & r2	{{r1}&&{r2;[*]}} | {{r2}&&{r1;[*]}}
r1 within r2	{[*];r1;[*]}&&{r2}

Table 9: SERE rewrite rules.

After the SERE rewrite rules are employed, PSL formulas can be reduced to eight SERE base-case formats, shown in Table 10 [72].

[*0]
b
{r}
r1 ; r2
r1 : r2
r1 | r2
r1 && r2
r[*]

Table 10: SERE base cases.

2. Automata Representation

After the PSL formula has been rewritten into a base-case format, it is converted into an equivalent automaton representation, as described next.

The construction of automata for accepting or rejecting input sequences based on temporal logic descriptions has been explored by a number of researchers. Alpern and Schneider showed how to translate temporal logic formulas into equivalent automata, to facilitate static proofs regarding a program's behavior [37]. Vardi also explored the connection between temporal logic and automata [73], as did Pnueli [46], who also developed a good deal of the temporal logic theory [58].

Much of the aforementioned research involves static behavioral proofs, and allows the possibility that an input sequence may be infinite in length. As such, Büchi automata, which permit infinite-length input sequences, are often used. For example, any LTL formula can be translated to an equivalent Büchi automaton [74]. In the context of dynamic verification, though, the input sequence is finite, and therefore Büchi automata are not necessary; instead, we can use a logic-based variant of classical automata, which consider only finite-length inputs.

a. Definitions

As pointed out by Boulé and Zilic [6], the other major difference between traditional automata and automata which accept those languages defined by temporal logic formulas is the input symbol alphabet. In classical automata, the input alphabet is mutually exclusive by definition; only one input symbol at a time can be a valid input condition, to the exclusion of all others. In automata based on propositional logic, the edges which define state transitions are boolean expressions, and more than one expression may evaluate simultaneously to *true*. Boulé and Zilic use the term "symbolic alphabet" to refer to the use of boolean expressions as automata edge conditions. In such a system, at each new clock cycle, the propositional variables are assigned input values, and the boolean expressions are evaluated. If an automaton is in a given state and has, on an outgoing edge, a boolean condition which evaluates to *true*, then the automaton may transition to the new state defined by that edge.

Supporting the automaton definition, we define sets of *propositional variables* and *boolean expressions*, an *input function* for the inputs (signal values), and a *logical evaluation function* (for evaluating the boolean expressions), as follows:

- P is a nonempty, finite set of *propositional variables,* $p \in P$. At each clock cycle, each propositional variable has an input value of *true* or *false*. Each propositional variable is an alphanumeric.
- β is a finite set of *boolean expressions*, $b \in \beta$, formed in the usual way from the elements of P, plus the symbols for conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$), plus parentheses.
- L is the *input function* $L : n \longrightarrow 2^P$, representing the current assignments of P at the nth clock cycle. The set of input assignment values in clock cycle n is denoted l_n. We refer to l_n as the nth "letter" of an input word. The clock cycle n is a non-negative integer.
- Φ is a *logical evaluation function*, over an assignment l ("letter") of the propositional variables in P, for boolean expressions in β:

$\Phi(b, l) \rightarrow \{true, false\}$. Φ evaluates propositional boolean formulas in the usual way. For example, if $b =$ "$x \wedge y$," and in clock cycle n, $l_n = \{x=false, y=true\}$, then we expect $\Phi(b, l_n) = false$.

We define each Propositional Logic Automaton (PLA)[10] as a five-tuple: A = {Q, q_0, L, F, δ}, where:

- Q is a nonempty, finite set of states.
- $q_0 \in Q$ is the *start state*.[11]
- L is the valuation function mentioned above, which provides input values to the automaton. Note that L will be the same for all the automata associated with a single hardware system being modeled.
- $F \subseteq Q$ is the set of accepting, or *final*, states.
- $\delta \subseteq Q \times \beta \times Q$ is the *transition relation* from state to state, via edges defined by boolean expressions. δ is a set of triples, { $(q,b,r) \mid q \in Q, b \in \beta, r \in Q$ }.

b. Computation and Acceptance

A state being described as *active* or *inactive* is a way of representing the current state of the *computation* of an input word. During computation, each individual state $q \in Q$ is described as either *active* (also representing true, or 1) or *inactive* (respectively false, or 0) during a clock cycle. The automaton initializes with all states *inactive* except the start state, q_0, which is *active*. Because they may be nondeterministic, it is permissible for more than one state in an automaton to be *active* simultaneously. The *active/inactive* characterization derives from the fact that our automata will be implemented in hardware designs as circuits, as described in Figure 44, with each state represented by a flip-flop. A state being *active* is analogous to a classical automaton being "in" a particular state during computation of some input, i.e. an *active* state represents the current computational state of the machine.

An input word is accepted by a PLA if and only if computation of the input word completes with one or more of the automaton's final states *active*. Note that acceptance will lag by one clock cycle—if an input word of length n is accepted, one or more final states will be *active* during clock cycle $n+1$.

The notion of being *active* or *inactive* is only meaningful during the *computation* of an input word, and is therefore not referenced in any of the automata construction algorithms in this chapter.

c. Transition Function

Next, we define how the transition function operates. An automaton may transition from state q to state r on a given clock cycle n if there exists an entry (q,b,r) in the set δ and $\Phi(b, l_n) = true$, where l_n is the set of true-false values of the propositional input variables during clock cycle n. If a transition to state r occurs, as a consequence of state q being *active* during clock cycle n and the input letter l_n permitting a transition to r, then state r will be *active* in clock cycle $n+1$. Each automaton state's *activity* is calculated,

[10] Here we differ slightly from Boulé and Zilic. Where they use intermediate symbols to abstract away underlying boolean expression operators, we model the boolean layer expressions, all the way down to raw alphanumeric variable names for actual HDL signals, plus AND, OR, and NOT.

[11] In classical nondeterministic finite automata, more than one start state is allowed. For our method, however, only a single start state is required.

on every clock cycle, independently, in this way. If we use the terminology *active*(*q*,*n*) to denote that state *q* is *active* during clock cycle *n*, Definition 1 reflects an automaton state's operation on some input word *v*.

$$\forall\, q,r \in Q,\, b \in B,\, l_n \in L,\, 0 \le n \le |v| : \quad active(r,n+1) \Longleftrightarrow \exists q,b : (active(q,n) \wedge \Phi(b,l_n) \wedge ((q,b,r) \in \delta)) \quad (1)$$

Henceforth, unless noted otherwise (e.g., "classical" automaton), all automata we describe are PLAs, and the terms "automaton" and "PLA" are used interchangeably in context.

d. Nondeterminism

PLAs may be either deterministic or nondeterministic. In classical automata, nondeterminism arises when the same input symbol is present on more than one outgoing edge from a state; in PLAs, nondeterminism arises when the outgoing edges from a state are not all pairwise logically exclusive. That is, if there exists at least one state in the automaton with more than one outgoing transition, to different successor states, that may be simultaneously satisfied by a single input assignment, the automaton is nondeterministic. Conversely, PLAs are deterministic if and only if, for every state, there exists no single propositional variable assignment which causes more than one outgoing edge's boolean expression to simultaneously evaluate to *true*, unless those edges all transition to the same successor state. Formally, a PLA is deterministic if it satisfies Definition 2, and is nondeterministic otherwise. A formal definition for determinism of a PLA is not given by Boulé and Zilic, so Definition 2 is our own.

$$\forall\, (q \in Q, \quad b_1,b_2 \in \beta, \quad r_1,r_2 \in Q, \quad l \in 2^P) :$$
$$\{\, (\, ((q,b_1,r_1) \in \delta) \wedge\ ((q,b_2,r_2) \in \delta) \wedge\ (\Phi(b_1,l) = true) \wedge\ (\Phi(b_2,l) = true)\,)\ \Rightarrow (r_1 = r_2)\,\} \quad (2)$$

e. Example

The following brief example illustrates some of the definitions. Suppose we have an automaton, *A*, which models a PSL formula. A diagram of *A* is shown in Figure 15.

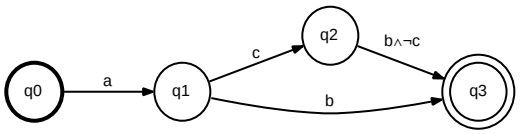


Figure 15: Example automaton *A*.

The automaton A is defined by the following elements:

- $Q = \{q_0, q_1, q_2, q_3\}$.
- $q_0 = q_0$.
- $F = \{q_3\}$.
- $\delta = \{(q_0, a, q_1), (q_1, b, q_3), (q_1, c, q_2), (q_2, b \wedge \neg c, q_3)\}$.
- The valuation function L, which provides input, is a representation of how the hardware circuits in the system behave over time. Suppose, for a particular initial state of the hardware, and inputs into the hardware, the signals a, b, and c take on the values, over time, shown in Figure 16. Each column indicates a clock cycle. For this example, we call the set of values an *input word*, namely v. The word v is comprised of five *letters*, l_0 through l_4, with each letter representing the set of signal values for that clock cycle. In this example, $v = l_0 l_1 l_2 l_3 l_4$ (an input *word* is composed of a series of input *letters*). A prefix of v is indicated with superscripts, e.g., $v^{0..3}$ for letter 0 through letter 3 of v.

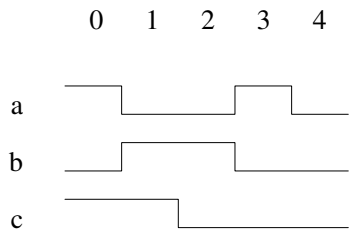
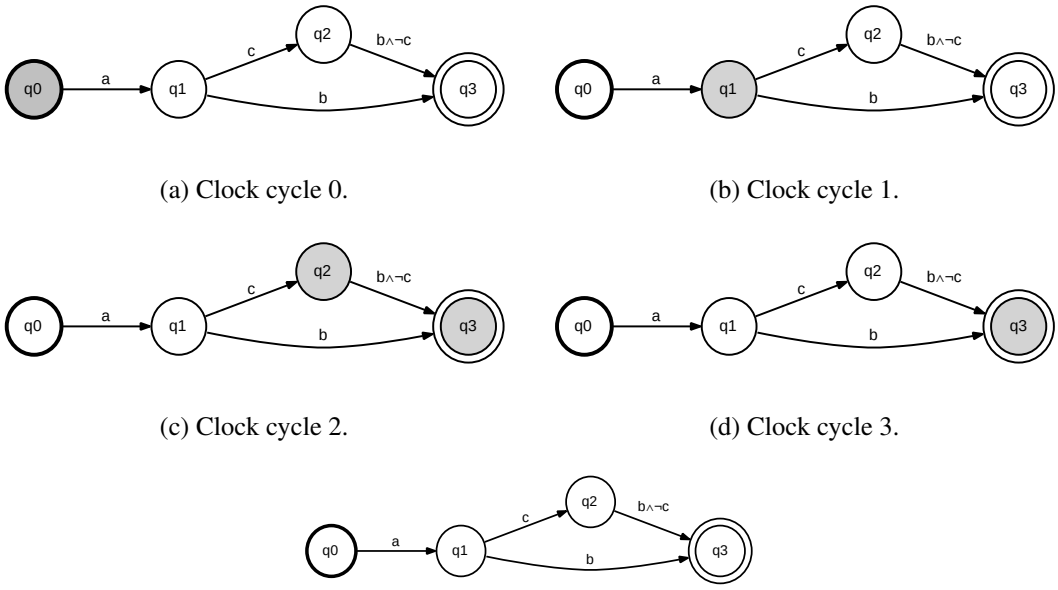


Figure 16: Example input word v.

In this example, l_0={a=*true*, b=*false*, c=*true*}, l_1={a=*false*, b=*true*, c=*true*}, l_2={a=*false*, b=*true*, c=*false*}, l_3={a=*true*, b=*false*, c=*false*}, and l_4={a=*false*, b=*false*, c=*false*}. Some researchers use a shorthand notation for representing the letters, in which a proposition (signal) is listed if it is *true*, and omitted if it is *false*, for that cycle. For example, using this shorthand notation, v={{a,c}, {b,c}, {b}, {a}, {}}. We will not use this shorthand notation, to avoid ambiguity. As an illustration why, examine clock cycle 2: representing l_2 by shorthand as simply the set {b}, implying (by the shorthand convention) that a and c are *false* during that cycle, might be confused with the boolean expression b, which by itself implies nothing about the values of a and c, which may be either *true* or *false* without affecting the evaluation of the boolean expression b.

The computation of input word v by automaton A proceeds as indicated in Figure 17. We shade an automaton state gray to indicate the state is *active* during that clock cycle.



(a) Clock cycle 0.

(b) Clock cycle 1.

(c) Clock cycle 2.

(d) Clock cycle 3.

(e) Clock cycle 4.

Figure 17: Computation of input word v on automaton A.

As noted above, indication of the acceptance of an input, by a final state becoming active, lags by one clock cycle. In the example, the final state q_3 is *active* only in clock cycles 2 and 3, indicating that the input word $v^{0..1}$ and the input word $v^{0..2}$ are accepted by the automaton, but not the input words v^0, $v^{0..3}$, or $v^{0..4}$ (the full length of v). We relate automaton acceptance and the PSL formal semantics in more detail in Chapter IX.

f. The Always *Operator*

In the absence of a PSL temporal operator like *always* or *never*, automaton computation of an input word will begin just once. In practice, most PSL assertions will use a temporal operator like *always*. In the circuit implementation of an automaton, the *always* operator is modeled by making the start state active on *every* clock cycle, rather than just the first clock cycle.

g. Some Differences from Boulé and Zilic Automata

Our automata differ slightly from those used by Boulé and Zilic. One difference is their use of "extended symbols," as a level of abstraction above primary symbols. For example (using the Verilog flavor of PSL), the boolean expression $(f \,||\, !g)$ would be modeled as a single "extended" symbol in their system, and replaced by a single letter, whereas it would be modeled directly as the boolean expression $(f \vee \neg g)$ in ours. Because the underlying disjunctions, conjunctions, and negations are not abstracted away in our system, we are able to perform many boolean simplifications that do not appear possible in their model [6]. Our boolean expressions are internally maintained in disjunctive normal form (DNF).

Another difference is the use of disjunction in the boolean expressions that define the automata edges. Boulé and Zilic do not use boolean disjunctions on edge conditions; instead, they model the disjunction implicitly, by adding another edge [6]. For example, if there is a transition from state q to state r by b_1 or by b_2, they would have an edge for b_1 and another edge for b_2. In our system, there would be a single edge represented by the expression $(b_1 \vee b_2)$. This can be slightly more effort for us to implement, but here again it allows us to perform boolean simplifications that their system does not appear to facilitate, resulting in fewer edges in many cases.

h. Conditional Mode and Obligation Mode

Boulé and Zilic use the concepts of *conditional-mode* automata and *obligation-mode* automata (also called *first-failure*, or *fail-mode* automata) to illustrate the difference between the semantics of accepting sequences and the semantics of accepting properties [71]. In PSL, sequences simply describe the occurrence of something. We say that a sequence "holds," or that an instance of it "is detected." On the other hand, properties in PSL express a temporal obligation, for which we detect *failures* to meet the obligation. For example, such obligations might be that the described event should occur *always*, occur *never*, occur *next*, occur *until* or *before* some other event, etc.

In PSL, SEREs are inherently in *conditional* mode, but properties are inherently in *obligation* mode [71]. As a result, the automata to accept *sequences* and the automata to accept *properties* may look quite different. In keeping with the method of Boulé and Zilic, we will denote *conditional-mode* automata, primarily used for SEREs (r), as $\mathrm{A}_C(r)$, and *fail-mode* automata, primarily used for properties (p), as $\mathrm{A}_F(p)$. Conditional-mode automata are covered first, in the section on SEREs, and fail-mode automata are covered later, in the section on properties. In our implementation, both conditional-mode automata and fail-mode automata are PLAs, as defined above.

3. Automata Operation for SEREs

a. Conditional Mode Automaton Checker Semantics, Defined

A checker employing a conditional-mode automaton accepts an input word v that terminates at clock cycle n if and only if at least one of the conditional-mode automaton's final states is *active* in clock cycle n+1, on computation of v.[12]

b. Empty Set, Empty Sequence, and Boolean Expressions

Where necessary, we can construct automata that accept no input sequences. This is accomplished by creating a single start state, with no edges and no final states. The language of the automaton is Ø.

We can also create an automaton that accepts only the empty input sequence. The automaton has a single start state, which is also an accepting state, and no edges. The language of the automaton contains only ε, the empty input sequence. For example, the PSL degenerate SERE "`[*0]`" accepts only the empty input sequence, by definition.

[12] Note the one clock-cycle computation delay, discussed earlier in this section.

In PSL, a boolean expression, by itself, has no temporal component. Therefore, constructing an automaton to accept it requires only two states, the start state and a final state, with a single edge from the start to the final state, defined by the boolean expression. The evaluation takes a single clock cycle, hence there is no need for a transition loop from the final state back to itself. If *b* is true in the single clock cycle, it accepts; otherwise, it does not accept.

Diagrams for each of these are given in Figure 18. Start states are in **bold**; accepting states are indicated by a double circle.

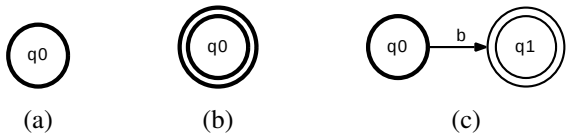


Figure 18: Automata for the empty set (a), the empty input sequence (b), and a boolean expression *b* (c).

The following subsections describe the algorithms for modifying or combining automata in various ways to accept new temporal sequences. Though we refer to acceptance of sequences here, some of the techniques can be applied to PSL properties as well, as discussed in the subsections that follow. Some of the techniques may introduce nondeterminism to an automaton; conversion of nondeterministic automata to their deterministic equivalents, when necessary, is also covered later. In some cases, the automata combination algorithms are similar to those used to implement operations on classical regular expressions (e.g., closure of an expression, concatenation of two expressions, and disjunction of two expressions).

c. Kleene Closure

To represent the Kleene closure [75] of an input sequence accepted by an automaton A, we duplicate all edges inbound to final states, and route the duplicated edges instead to the start state. When the automaton accepts an input sequence, it is simultaneously returned to the start state. In addition, since Kleene closure includes zero or more instances of a sequence, if the start state in A was not an accepting state, we make it into an accepting state, to accept the empty input sequence. See Figure 19 for an example.

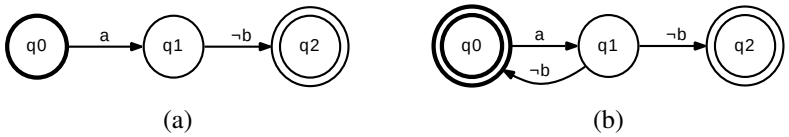


Figure 19: Closure example: automata for accepting a sequence (a), and its Kleene closure (b).

d. Concatenation

For sequence concatenation, we connect the left-hand automaton $A_C(left)$ and the right-hand automaton $A_C(right)$ as follows. For every final state in $A_C(left)$, add an outgoing edge that matches each outgoing edge from the start state in $A_C(right)$. An example is shown in Figure 20.

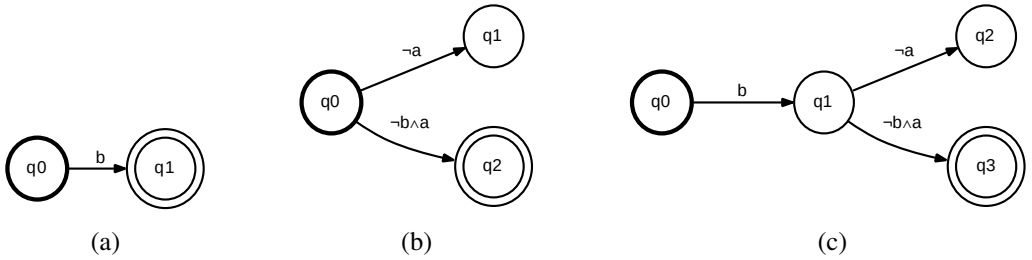


Figure 20: Concatenation example: automata for accepting a lefthand sequence L (a), a right-hand sequence R (b), and the concatenated sequence L ; R (c).

e. Fusion

Fusion is very similar to concatenation, but has a cycle of overlap. In sequence fusion, the final clock cycle of the left-hand sequence holds simultaneously with the first cycle of the right-hand sequence. Given a left-hand automaton $A_C(left)$ and a right-hand automaton $A_C(right)$, we fuse them by merging the incoming edges of the final states in $A_C(left)$ with the outgoing edges of the start state in $A_C(right)$. The edges are combined by conjunction. If a newly-formed edge condition simplifies to *false*, the edge is omitted.

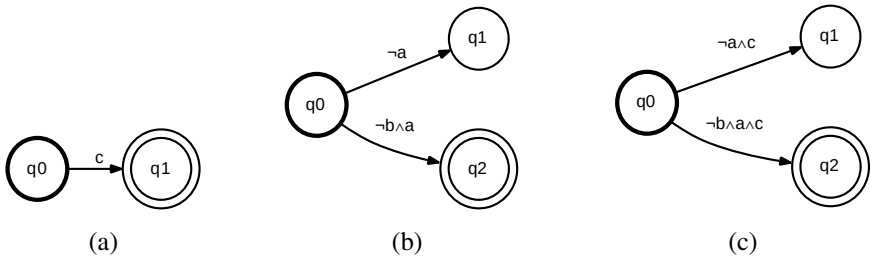


Figure 21: Fusion example: automata for accepting a lefthand sequence L (a), a right-hand sequence R (b), and the fused sequence L : R (c).

f. Disjunction

Disjunction of automata is straightforward. Given two input automata $A_C(m)$ for sequence m and $A_C(n)$ for sequence n, we create a new automaton $A_C(m|n)$ for sequence $m \mid n$ by combining the start states of $A_C(m)$ and $A_C(n)$ into a single new start state, with all the other states and edges unchanged. See Figure 22.

g. Length-Matching Intersection

Length-matching intersection, indicated by connecting two sequences with the `&&` operator, means that the new sequence holds if the two original sequences both hold over the same number of clock cycles. The method is similar to the *product construction*, used to compute the intersection of regular languages [75]. The idea behind the product construction is to generate state *pairs*, using one state at a time from each input automaton, and see what each individual automaton would do on a given input. For example,

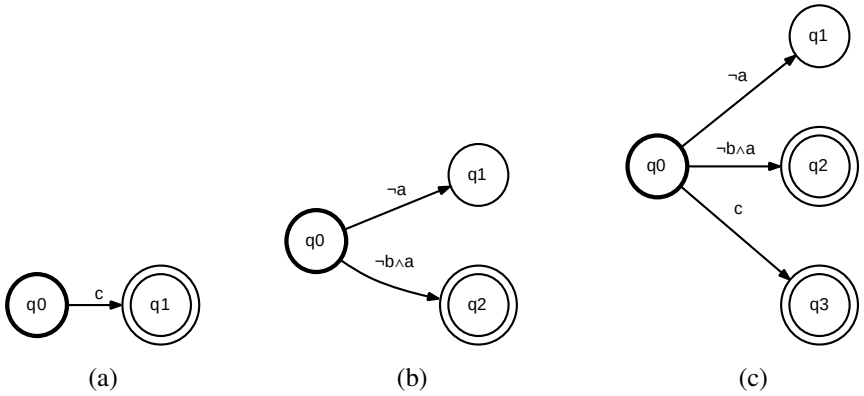


Figure 22: Disjunction example: automata for sequence *m* (a), automata for sequence *n* (b), and the automata for sequence *m* | *n* (c).

suppose we have classical automata $A_C(m)$ and $A_C(n)$. If $A_C(m)$ is in state *p* and $A_C(n)$ is in state *q*, then we construct the state pair (*p,q*) in the new automaton $A_C(m\&\&n)$, and examine its possible transitions, beginning with the (*p,q*) pair representing the start states. Suppose on input symbol *x* that $A_C(m)$ moves from state *p* to state *r* and A(*n*) moves from state *q* to state *s*. Then, we construct new state pair (*r,s*) in $A_C(m\&\&n)$, and make a transition in $A_C(m\&\&n)$ from state (*p,q*) to state (*r,s*). Each time we consider a state pair like (*p,q*), we need to consider all the possible inputs, generate any newly reachable state pairs like (*r,s*), maintaining the reachable states in $A_C(m\&\&n)$ on a stack. By keeping only the reachable states on a stack, we never need to consider the unreachable states. If the input automata have *j* and *k* states, respectively, the intersection automaton could have as many as $j \times k$ states, but will usually have less. By considering only the reachable states, we perform a minimum of work in the algorithm. We continue until the stack is empty.

When using PLAs, there is an additional complication. With classical automata, when considering the outgoing transitions for a state pair like (*p,q*) we only needed to consider the possible input symbols, whose size would normally be denoted |Σ|. In a PLA, though, we need to consider all logical combinations of the boolean expressions which contribute to the outgoing edges of *p* and *q*. For example, if *p* has *j* outgoing transitions and *q* has *k* outgoing transitions, we have to consider $j \times k$ total cases, where each case is the conjunction (logical and) of an outgoing boolean condition from *p* with an outgoing boolean condition from *q*.

In both classical and propositional-logic-based automata, a state pair (*p,q*) is marked as accepting in the combined automaton only if state *p* was accepting in $A_C(m)$ and state *q* was accepting in $A_C(n)$.

Consider the example in Figure 23. Starting with state pair (*q0,q3*), we must consider four possible output conditions, since *q0* and *q3* each have two outgoing edges in the original automata. These conditions are $a \wedge \neg b$, $a \wedge c$, $b \wedge \neg b$, and $b \wedge c$. Since $b \wedge \neg b$ simplifies to *false*, this combination does not need to be considered. In the resulting automaton, only state (*q2,q5*) is accepting.

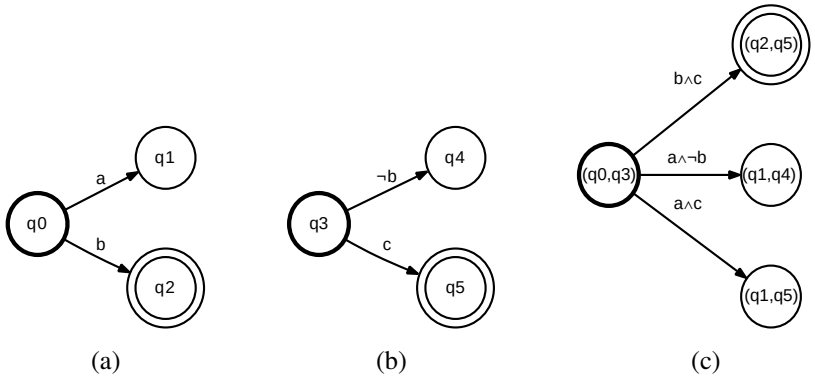


Figure 23: Length-matching intersection example: automata for sequence *m* (a), automata for sequence *n* (b), and the automata for sequence *m* && *n* (c).

h. SERE Base Cases, Summary

Now that we have described the automata generation methods, let us revisit the "base cases" for interpreting SEREs and Properties. The SERE bases cases are listed again in Table 11 [72], along with their respective automata-generation procedure.

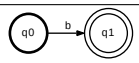
SERE Base Case	Automata Implementation
Any formula accepting Ø	
`[*0]` (empty string only)	
`b`	
`{r}`	$A_C(r)$
`r[*]`	Kleene closure of $A_C(r)$ (Fig. 19)
`r1 ; r2`	Sequence concatenation of $A_C(r1)$ and $A_C(r2)$ (Fig. 20)
`r1 : r2`	Sequence fusion of $A_C(r1)$ and $A_C(r2)$ (Fig. 21)
`r1 | r2`	Sequence disjunction of $A_C(r1)$ and $A_C(r2)$ (Fig. 22)
`r1 && r2`	Length-matching sequence conjunction of $A_C(r1)$ and $A_C(r2)$ (Fig. 23)

Table 11: SERE base cases, with implementation strategies.

4. Automata Operation for Properties

a. Fail Mode

As mentioned earlier, we need to be able to detect the occurrence of sequences with conditional mode automata A_C, and also detect the failure of sequences to occur, with fail-mode automata, denoted A_F. Conditional-mode automata may be nondeterministic or deterministic, because implementing nondeterministic checker automata in hardware circuits is straightforward (discussed later in this chapter). However, fail-mode mode automata must be deterministic, because *every* activation of the start state obligates the sequence to occur, else the implication fails. The process for converting a conditional-mode automaton for a sequence into a fail-mode automaton is given in Algorithm VII.1 [71].

Algorithm VII.1 Conversion of automata to failure-detection mode.

1. CONVERT_TO_FAIL_MODE(A):
2. Determinize A
3. Add a state called *fail* to A
4. For each state *s* in A do:
5. For each boolean assignment of the propositional input variables not covered by an edge in A:
6. With the "unused" boolean assignment, create a new edge in δ from state *s* to state *fail*
7. Remove any edges from state *s* which transition to a final state
8. Mark state *fail* as the new final state
9. Return A

b. Fail Mode Automaton Checker Semantics, Defined

A checker employing a fail-mode automaton accepts an input word that terminates at clock cycle *n* if and only if the fail-mode automaton's *fail* state is *never* active, from the beginning of the input word until clock cycle *n*+1, inclusive. Consequently, if an input word *v* causes the *fail* state to be visited, any input word formed by adding any suffix to *v* will not be accepted by the checker. We observe, incidentally, that this interpretation causes a fail-mode checker automaton to accept *prefix-closed* sets (of input words), in the sense described by Lamport [76] and Schneider [34].

c. Failure Reporting

Note, however, that it is useful, in terms of failure reporting, to have the checker signal a "fail" output only when the automaton's *fail* state is visited, rather than continuously output a "fail" thereafter. By sending a "fail" output from the checker only when the automaton's *fail* state is visited, we can observe when multiple failure-inducing events are present, count them, and easily locate them all later for analysis. This is the technique described by Boulé and Zilic for their implementation, and we use it as well [6]. After a simulation run, if a checker employs a fail-mode automaton, we assess an input sequence as failed if the checker output signal *ever* outputs a "fail," even for one clock cycle; however, we are able to locate multiple failure instances (if more than one exists) for a simulation run, by observing the entirety of the checker's output signal.

d. Determinization

In step 2 of the fail-mode conversion algorithm, the automaton must be converted to deterministic form, discussed next.

A classical NFA can be converted to a DFA using a process called *determinization*. This is normally done using the *subset construction*, as described by Hopcroft, Motwani, and Ullman [75]. As ob-

served by Boulé and Zilic, when using logic-based automata, determinization introduces some complications not encountered in the classical case.

First, a brief description of the classical subset construction algorithm. In converting a nondeterministic automaton N into a deterministic automaton D, the set of possible states in D is represented by the power set of states in N; each possible state in D is represented as a *subset* from the power set of Q_N. Since the maximum possible size of Q_D is $2^{|Q_N|}$ but in practice is usually much smaller, it is more efficient to use a "lazy evaluation." Under lazy evaluation, only the known reachable states of D are considered.

The determinization procedure we use for PLAs is listed in Algorithm VII.2. The main difference between the classical algorithm and the propositional-logic-based algorithm is in steps 8-9. In the traditional algorithm, we only need to consider each possible symbol from the input alphabet Σ to determine the set of states reachable from the current state, s. In the logic-based variant, we need to consider each condition in the boolean powerset (all possible combinations of *true-false* values) of propositional variables whose values determine outbound transitions from s; this is necessary in order to preserve determinism in the resulting automaton, D.

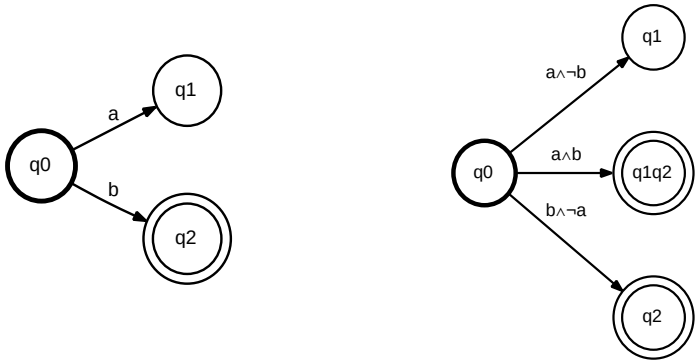
Algorithm VII.2 Determinization algorithm for logic-based automata.

1. DETERMINIZE(N):
2. $S = \emptyset$. *Visited* = $\emptyset$. (S is the set of currently known reachable states in D.)
3. Push q_0 from N onto S.
4. While $S \neq \emptyset$:
5. s = Pop(S)
6. if $s \notin$ Visited:
7. Push(s, *Visited*)
8. Compute the set *out_s,* of propositional variables needed to compute outgoing edges of s.
9. Compute the boolean power set *power_s*, of the variables in *out_s.*
10. For each condition *cond* in *power_s*:
11. If there are any available outgoing transitions in N from state s under the assignments in *cond*, compute a new deterministic state *next_s*, formed as the set of states that were reachable in N, via *cond*, from s. If there are no outgoing transitions in N from s that are logically *true* under the assignments in *cond*, do not create *next_s*, and exit the loop back to step 10.
12. If *next_s* $\notin S$: Push(S, *next_s*)
13. Add the state *next_s* to D (if not already created), and add edge (*s, cond, next_s*) to D
14. For all states $q \in$ D:
15. If one or more of the substates of q was a final state in N:
16. Mark q as a final state in D.
17. Return D.

Though created independently, our PLA determinization algorithm appears similar to what Boulé and Zilic refer to as "strong determinization" in their text [6]. However, the algorithm they refer to as "weak determinization" is not necessary in our implementation, which always manipulates boolean expressions at the level of primary symbols (whereas they describe using an intermediate, alternate symbol representation in some cases).

e. Determinization Example

Consider the automaton N in Figure 24. If interpreted as a classical automaton, it would be deterministic. Interpreted as a PLA, however, it is not. In the following steps, we describe the operation of the determinization algorithm on N, as it is converted to a deterministic equivalent, D.



(a) Nondeterministic PLA *N*. (b) Deterministic PLA *D*.

Figure 24: Determinization example.

Suppose that only the start state, q_0, has been pushed onto the stack, *S*, as in line 3. State q_0 is popped from *S* in line 5, and placed into *s*. The *Visited* queue is empty, and we push q_0 onto *Visited* in line 7. In line 8, we compute *s_out*, the set of propositional variables used in the boolean expressions on outgoing edges of q_0, so *s_out* = {*a, b*}. In line 9, we compute the boolean powerset of *s_out*. Note the *boolean powerset* differs semantically from the traditional set-theoretic powerset; with the boolean powerset, we are creating an explicit representation of the boolean powerset alphabet, in this case *cond* = $\{\neg a \wedge \neg b, \neg a \wedge b, a \wedge \neg b, a \wedge b, \}$. For each element (boolean transition condition) of the set *cond*, we execute lines 11-13.

The first condition, $\neg a \wedge \neg b$, does not yield a reachable successor state in *N* from q_0, so lines 12-13 take no action. The second condition, $\neg a \wedge b$, leads to the creation of state q_2 in *D*, and an edge to it. The third condition, $a \wedge \neg b$, leads to the creation of state q_1 in *D* and an edge to it. The fourth condition, $a \wedge b$, yields the reachable states q_1 and q_2 in *N*, leading to the creation of state q_1q_2 in *D*. States q_1, q_2, and q_1q_2 are pushed onto *S*, but they have no outgoing edges in *N*, so no more states are added to *D*.

f. Maintenance of Determinism by the Algorithm

Theorem 7.1: Algorithm VII.2 produces only deterministic propositional logic automata.

Proof: On the structure of the PLA D, as it is constructed by the algorithm.

Invariant: Suppose D is a deterministic PLA being computed by the algorithm. No action of the algorithm, i.e. the addition of new states and edges, causes D to become nondeterministic.

Initial Condition: D is initially an empty PLA (no states and no transitions), which vacuously obeys our definition of determinism.

Maintenance: First observe that the states and transitions of D are added only in the loop at steps 10-13 of the algorithm; specifically, at step 13. Suppose, in an iteration of the loop for steps 10-13 in the algorithm: variable *s* is assigned to some state in D, and *cond* is a boolean condition, from the boolean powerset *power_s,* under consideration. The state *s* represents a set of states from N, which we denote s_n.

- First, suppose state *s* currently has no outgoing transitions in D.
  - Suppose state *s* gains no successor from steps 10-13, because *cond* yielded no outgoing transitions from any of s_n in N. Since no new states or edges are added to D, determinism is preserved.
  - Suppose one or more of s_n in N has an outgoing transition available on *cond*. The set of successor states from N forms the set called *next_s*, which is the single successor state added to D in line 13. Now, a single edge (*s, cond, next_s*) is added to D. Because *s* had no outgoing transitions in D before, and now has a single outgoing transition and a single successor state in D, determinism is preserved.
- Next, suppose state *s* currently has one more more outgoing transitions in D.
  - Suppose state *s* gains no successor from steps 10-13, because *cond* yielded no outgoing transitions from any of s_n in N. Since no new states or edges are added to D, determinism is preserved.
  - Suppose one or more of s_n in N had an outgoing transition available on *cond*. The set of successor states from N forms the set called *next_s*, which is the successor state added to D in line 13 (if it is not already in D). The edge (*s, cond, next_s*) is also added to D in step 13.
    - Suppose that D already has an outgoing edge from s, but the addition of (*s, cond, next_s*) does not create nondeterminism in D. The invariant holds.
    - Suppose D already has an outgoing edge from *s* such that the addition of (*s, cond, next_s*) violates determinism; the pre-existing edge would have to have been created on some previous iteration of the loop for steps 10-13. Denote this pre-existing edge (*s, prev_cond, next_s1*) and the current (to be added) edge (*s, current_cond, next_s*2), with *next_s1* ≠ *next_s2*. In this case, by the determinism definition, we have $s = q$, *prev_cond* = b_1, *current_cond* = b_2, *next_s1* = r_1, *next_s2* = r_2, and l is an assignment of *cond* under which both b_1 and b_2 are true (see the determinism definition in Equation 2). This leads to a contradiction, however: if b_1 and b_2 are both satisfied by the same assignment of *cond*, then the sub-states of N forming *next_s1* and *next_s2* should all have been added together, the first time that assignment of *cond* occurred, and there should be no current members in *next_s2*; no two unique assignments of *cond,* satisfying all our assumptions, can exist.
      - Our assumption that the new edge yields nondeterminism produces a contradiction. Therefore, the invariant must hold.
- We conclude that, under all conditions in which states and edges are added to D, the operation of the algorithm preserves determinism.

g. Property Base Cases

Several of the Property base cases can be simplified further from Table 8. Since our representation employs full boolean expressions rather than intermediate symbols, as those used by Boulé and Zilic, we can simplify "`!b`"[13] and "`b↔b`"; they both resolve to further boolean expressions, represented by just "`b`."

[13]The expression "`!b`" is the negation of "`b`," in the Verilog flavor of PSL. If using VHDL, we would write it as "`not b`".

For a property that consists of only a boolean expression "`b`," the automaton for the property is constructed by converting the conditional-mode automaton for "`b`" to a fail-mode automaton, as described above (See Figure 25). Implementing a SERE as a property is performed in the same manner: take the sequence's conditional-mode automaton, and convert it to a fail-mode automaton. Additionally, if the SERE is *strong* we add a transition from every state in the fail-mode automaton, to the *fail* state, on detection of the end-of-execution signal. That way, any "in-flight" sequences which have not completed at the end of execution (but were obligated to do so, due to the strong operator), cause a property failure.

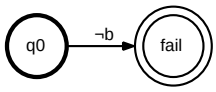


Figure 25: Boolean b, interpreted as a property.

The disjunction of properties, as in "`p1 || p2`," was handled by a rewrite rule, since it is non-primitive (in the Simple Subset). The conjunction of properties, however, is a base case. Note that, unfortunately, in the Verilog flavor of PSL, the token "`&&`" is used both for length-matching SERE conjunction and for property conjunction, though the semantics are quite different. With SEREs, we used conditional-mode automata and length-matching of sequences over multiple cycles; with properties, we use fail-mode automata which are evaluated for failure detection at every clock cycle. Given a property conjunction "`p1 && p2`," we first construct fail-mode automata for "`p1`" and "`p2`," respectively, and then use the automata *disjunction* algorithm, which was introduced above for SERE disjunction. The reason this works semantically is that it is similar to an application of DeMorgan's Laws: $\neg(p1 \wedge p2) = (\neg p1 \vee \neg p2)$. Essentially, the property "`p1 && p2`" is for detecting a *failure* of "`p1 && p2`," which occurs if either "`p1`" *or* "`p2`" fails.

The PSL "`abort`" operator acts like a reset, terminating any sequence obligations currently "in flight." It is handled as follows. Given a property "`p abort b`," first construct the fail-mode automaton for "`p`," or $A_F(p)$. Next, modify every edge in $A_F(p)$ by adding "$\wedge\neg b$" to its transition condition. Essentially, the occurrence of "`b`" invalidates all possible transitions, causing a "reset" of the automaton.

h. Suffix Implication

The last property base case is suffix implication ($\mapsto$). In Chapter VI, we described both of PSL's suffix implication operators, $\mapsto$ (overlapping suffix implication) and $\Mapsto$ (non-overlapping suffix implication). Because of the final rewrite rule in Table 7, the non-overlapping operator $\Mapsto$ gets rewritten in terms of the overlapping operator $\mapsto$, so the latter is the only suffix implication base case that needs to be handled.

We mentioned in Chapter VI that the semantics of suffix implication span multiple clock cycles, unlike boolean implication ($\rightarrow$), which is evaluated in a single clock cycle. Informally, the semantics of suffix implication ($\mapsto$) dictate that, starting in the clock cycle when the left-hand sequence is observed (completes), the right-hand sequence *must* be observed, else the overall assertion fails.

Boulé and Zilic devised the following strategy for creating an obligation-mode automaton for suffix implication:

- Given an automaton for accepting the right-hand sequence, convert it into a fail-mode automaton, which detects a failure of the right-hand sequence to occur.
- Using the automata sequence-fusion algorithm, fuse the conditional-mode automaton for the left-hand sequence with the obligation-mode automaton for the right-hand sequence.

Figure 26 shows an example of suffix implication. Consider the PSL formula "`assert always ({a} ↦ {c;d;e})`." The sequence represented by the boolean expression "`a`" is accepted by the conditional-mode automaton in (a). The sequence represented by the SERE "`{c;d;e}`" is accepted by the automaton in (b). The automaton in (b) is already deterministic, for simplicity. In (c), the conditional-mode automaton from (b) has been converted into a fail-mode automaton, in accordance with Algorithm VII.1. Finally, in (d), the automata from (a) and (c) have been fused, using the sequence fusion algorithm. When "`a`" is observed to be true, it triggers the obligation that "`c`" is true in the same clock cycle, followed by "`d`" then "`e`" in the following clock cycles. If "`a`" is not observed to be true, then no obligation is incurred, and the assertion vacuously holds in that instance. During this process, we started with conditional-mode automata for each side of the implication, converted the right-side automata to fail mode, and ended up with a fused fail-mode automaton for accepting the suffix implication, which is a property.

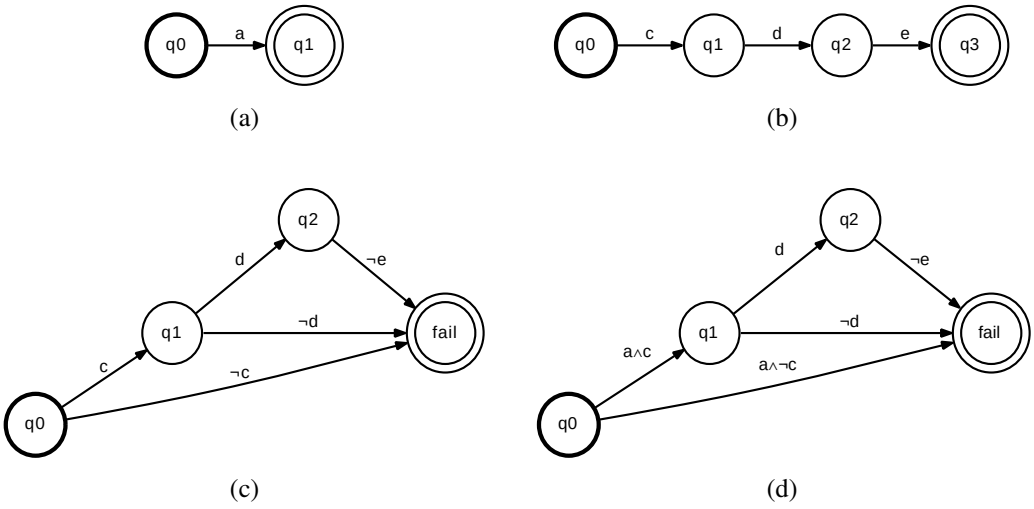


Figure 26: Suffix implication example.

i. Property Base Cases, Summary

The implementation strategies for the property base cases are listed in Table 12.

Property Base Case	Automata Implementation
`b`	$A_F(b)$
`r`	$A_F(r)$
`r!`	$A_F(s)$, add end-of-execution marker and transitions
`(p)`	$A_F(p)$
`p1 && p2`	$A_F(p1) \mid A_F(p2)$
`p abort b`	$A_F(p)$, add "$\wedge\neg b$" to edges
`r ↦ p`	implement as $A_C(r) : A_F(p)$ (Fig. 26)

Table 12: Property base cases, with minor simplifications from Table 8, and implementation strategies.

5. DFA Minimization

Once we have a PSL formula in its automaton representation, we apply some simple minimization, pruning unreachable states and dead states (those which can never lead to a final state). If the automaton is deterministic, we use the following procedure to minimize it further, for efficiency.

Hopcroft, Motwani, and Ullman also provide our reference algorithm for DFA state minimization. Classical DFAs can be efficiently minimized; in the case of NFAs, however, there is no published algorithm which guarantees an efficient (sub-exponential) minimization, according to Hopcroft [75]; Jiang and Ravikumar show the general problem of NFA minimization to be PSPACE hard [77]. As mentioned previously, converting an NFA to a DFA can result in an exponential increase in the number of states. With logic-based automata, though, converting an NFA to a DFA and then minimizing the DFA will usually be reasonably efficient. We have only implemented a minimization algorithm for DFAs.

As was the case with determinization, the algorithm for PLA automata differs slightly from the classical algorithm. The key idea behind the classical DFA minimization algorithm is to identify which pairs of states are and are not distinguishable, and record them in a distinguishability matrix. If two states are found to be distinguishable (not equivalent), the entry for that pair is marked with an "X" in the matrix. Distinguishability is transitive. For example, consider the transition from states p and q on input symbol a: if $\delta(p,a) = r$ and $\delta(q,a) = s$ and states r and s are distinguishable, then states p and q are also distinguishable. By definition, all final states are distinguishable from non-final states, and vice-versa. After the matrix has been filled, state pairs that do not have an "X" (are not distinguishable from each other) are considered equivalent, and the states will be merged. The steps are listed in Algorithm VII.3.

The difference between the "table-filling" portion of the classical algorithm, and the "table-filling" for PLAs is in lines 9-12. In the classical algorithm, from state pair (p,q), we only need to consider any possible input symbol a from the input alphabet Σ to determine the possible successor state pairs (r,s). In logic-based automata, on the other hand, we need to consider all possible *conjunctions* of the outgoing conditions for p and the outgoing conditions for q, which would allow (r,s) to be reached under the same conditions.

Algorithm VII.3 Minimization algorithm for propositional logic automata.

1. MINIMIZE_DFA(D):
2. Trim any unreachable states and dead states from D (dead states can never lead to an accepting state).
3. If all states are non-final or all states are final, then there will be no distinguishable pairs. Return D.
4. Create a *distinguishability* matrix M, of |Q|× |Q| rows and columns.
5. For each state pair (*p,q*) in Q:
6. If *p* is final and *q* is not, or if *q* is final and *p* is not, mark (*p,q*) with an "X".
7. For *p* in nonfinal states (Q - F):
8. For *q* in nonfinal states (Q - F), $p \neq q$:
9. *conditions* = Ø. (compute the set of conjunctions of outgoing conditions from *p* and *q*)
10. for d_p in outgoing(*p*):
11. for d_q in outgoing(*q*):
12. construct $cond = d_p \wedge d_q$, simplify if needed. If $cond \neq false$, add it to *conditions*.
13. For *cond* in *conditions*:
14. Compute next states *r, s*, where (*p,cond,r*) ∈ δ and (*q,cond,*s) ∈ δ.
15. If M[r,s] == "X," then mark M[p,q] = "X."
16. Otherwise, if r ≠ s:
17. Put (*p,q*) on a "taglist" for (*r,s*): if (*r,s*) later gets an "X," then (*p,q*) gets one too.
18. After the matrix M has been filled, merge the equivalent states.
19. Return D.

Though Boulé and Zilic do eliminate dead states and unreachable states, they do not appear to implement a full DFA minimization algorithm like ours [6].

6. Automata Conversion to HDL

Once we have an automaton to accept or reject input sequences in accordance with a PSL assertion, translating the automaton into a synthesizable HDL representation is not difficult. As we have seen, some of the automata will be nondeterministic, and some are necessarily converted to deterministic form; in circuit-form representation, though, either is acceptable. In a circuit representation of an automaton, each state is a flip-flop, and can be either active or inactive. On each clock cycle, each state may transition outbound to zero or more of its adjacent states, depending on whether the boolean expressions on its edges evaluate to *true* during that clock cycle. A state is *entered* on a given clock cycle if any of the state's *inbound* transitions is taken from an active state. More formally, a state *q* is active in clock cycle *n+1* if there exists at least one state *p* that was active in clock cycle n, a transition (*p,b,q*) exists in δ, and $\Phi(b,\text{n})$ evaluates to *true* (expression *b* is

true during clock cycle n). The transitions into q form an implicit disjunction (if one or more transitions into q can be taken, q becomes active).

Accepting states indicate an observed, or "hold," condition in conditional-mode automata, and a "fail" condition in fail-mode automata. In either case, the output of the checker is the disjunction of the accepting states: if one or more accepting states is active, the *hold* or *fail* output signal of the checker is triggered.

Both logic-based NFAs and DFAs may be *pipelined*. That is, they can have the start state go active during consecutive clock cycles, so that more than one observed "instance" of a sequence is being processed at once. In the case of fail-mode DFAs, the deterministic requirement guarantees that, once initiated, a given obligated "instance" either reaches a final state (triggers a "fail" signal) or ceases to exist and "falls off the automaton" (fails to fail) if it ever has no available transition to make during a clock cycle. Equivalently, a "sink" state can be implemented, so that all DFA transitions are fully specified, but this is not required.

Hardware design languages like Verilog and VHDL contain constructs for blocking assignments, which operate more like assignments in an imperative language, and non-blocking assignments, which model assignments made collectively and in parallel. Non-blocking assignments can occur, for example, inside a Verilog *always* block. In such a block, when the block's sensitivity list is triggered by a change, such as a clock cycle, the non-blocking assignments are evaluated in and assigned in parallel; the right-hand side of the assignments are computed using the last clock cycle's values, and then the left-hand sides of the assignments are given their new values in parallel. In the case of Verilog, this ability to maintain a "state" value is expressed by the *reg* construct, as opposed to the *wire* construct. This is the functionality we desire in modeling an automaton, where each state of the automaton is updated independently on every clock cycle, and becomes active or inactive based on the preceding clock cycle's values.

Consider an example, called SERE3. The PSL formula, rewritten formula, and automaton computed by our checker generator are shown in Figure 27. The Verilog code output by our checker-generator is in Figure 28. The state called "fail" in the automaton is represented by the state "SERE3_q4" in the Verilog code.

assert always ({a} |=> ({{b}[*3]; c})) @ (rose(clk));
assert {[+]} |-> ({{a} ; true} |-> ({{b} ; {b} ; {b} ; c})) @ (rose (clk)) ;

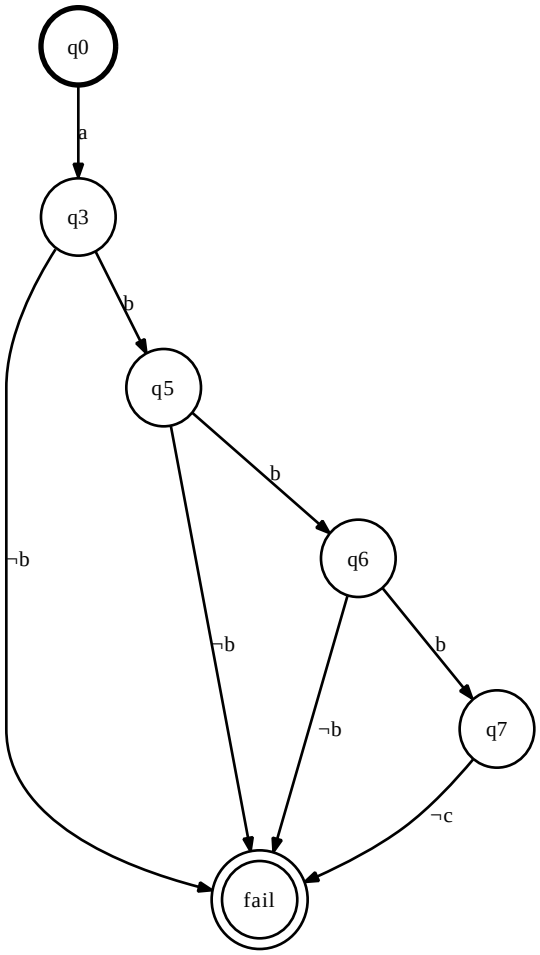


Figure 27: Checker automaton example.

```
module SERE3 (clk, reset, c, b, a, holds);
  input clk;
  input reset;
  input c;
  input b;
  input a;
  output holds;
  reg SERE3_q0;
  reg SERE3_q3;
  reg SERE3_q5;
  reg SERE3_q4;
  reg SERE3_q7;
  reg SERE3_q6;
  // Original assertion = assert always ({a} |=> ({{b}[*3]; c})) @ (rose(clk));
  // Rewritten assertion = assert {[+]} |-> ({{a} ; true} |-> ({{b} ; {b} ; {b} ; c})) @ (rose (clk)) ;
always @(posedge clk, posedge reset)
begin
  if (reset) begin
    SERE3_q0 <= 0;
    SERE3_q3 <= 0;
    SERE3_q5 <= 0;
    SERE3_q4 <= 0;
    SERE3_q7 <= 0;
    SERE3_q6 <= 0;
  end
  else if (clk) begin
    SERE3_q0 <= 1;
    SERE3_q3 <= (SERE3_q0 && ((a) === 1));
    SERE3_q5 <= (SERE3_q3 && ((b) === 1));
    SERE3_q4 <= (SERE3_q5 && ((b) !== 1)) || (SERE3_q6 && ((b) !== 1)) || (SERE3_q3 && ((b) !==
1)) || (SERE3_q7 && ((c) !== 1));
    SERE3_q7 <= (SERE3_q6 && ((b) === 1)); SERE3_q6 <= (SERE3_q5 && ((b) === 1));
  end
end
assign holds = ~(SERE3_q4 ); // Holds when fail state not visited
endmodule
```

Figure 28: Verilog example: automatically generated for SERE3 automaton.

In the SERE3 example, note that we use the Verilog expression "=== 1" to test if a signal is high, and the expression "!== 1" to test if the signal is not high. This illustrates an important issue regarding assertions and hardware description languages.

Input values in a real, physical system are not always high or low ("1" or "0") at the physical level. Electrically, they can also be modeled as unknown ("X"), high impedance ("Z"), etc. In evaluating an assertion, though, it is unsatisfactory to say that the assertion "neither holds nor fails." We would like its output value to always be defined. In order to ensure this, our checker-generator uses the test-for-equality expressions above in the Verilog case, since they always return a "0" or a "1." There are a great many equality and inequality operators in Verilog, and some may return other values like "X" or "Z" (instead of "0" or "1"), which do not seem as suitable for use in modeling assertions [61].

Another issue that should be addressed in the context of hardware assertions is that the underlying boolean expressions in a hardware design language can be complex. For example, Verilog has many unary and binary logical and bitwise operators [61] that we have not yet implemented with the basic boolean expressions; doing so completely requires a full front-end parser for each flavor (VHDL, Verilog, etc.) of PSL in use.

D. TOOL DESCRIPTION: PSL2HDL

To implement the checker-generator method of Boulé and Zilic, with minor modifications, we wrote a software tool in Python, which we call psl2hdl. The lexer, parser, and automata definition are slightly modified from those used by Findenig [4], but the rest of the software was originated for this research. The project totals just under 8,000 lines of code.

The parser is based on the Python 3rd-party plug-in called PLY, by Baez [78]. PLY, in turn, is based on the methods used in the lex-yacc family of parsers [79]. PLY is an LR-parser. Our tool parses PSL's Foundation Language, which is a superset of the PSL Simple Subset [56]. We implement abstract syntax trees ("parse trees"), using a custom-defined Python type called *psl_tree*. We also create a tool for exporting the parse trees to the DOT-language format, so they can be displayed using programs like Graphviz [80]. An example, created by our tool, is in Figure 29.

assert always ({a} |=> ({{b}[*3]; c})) @ (rose(clk));

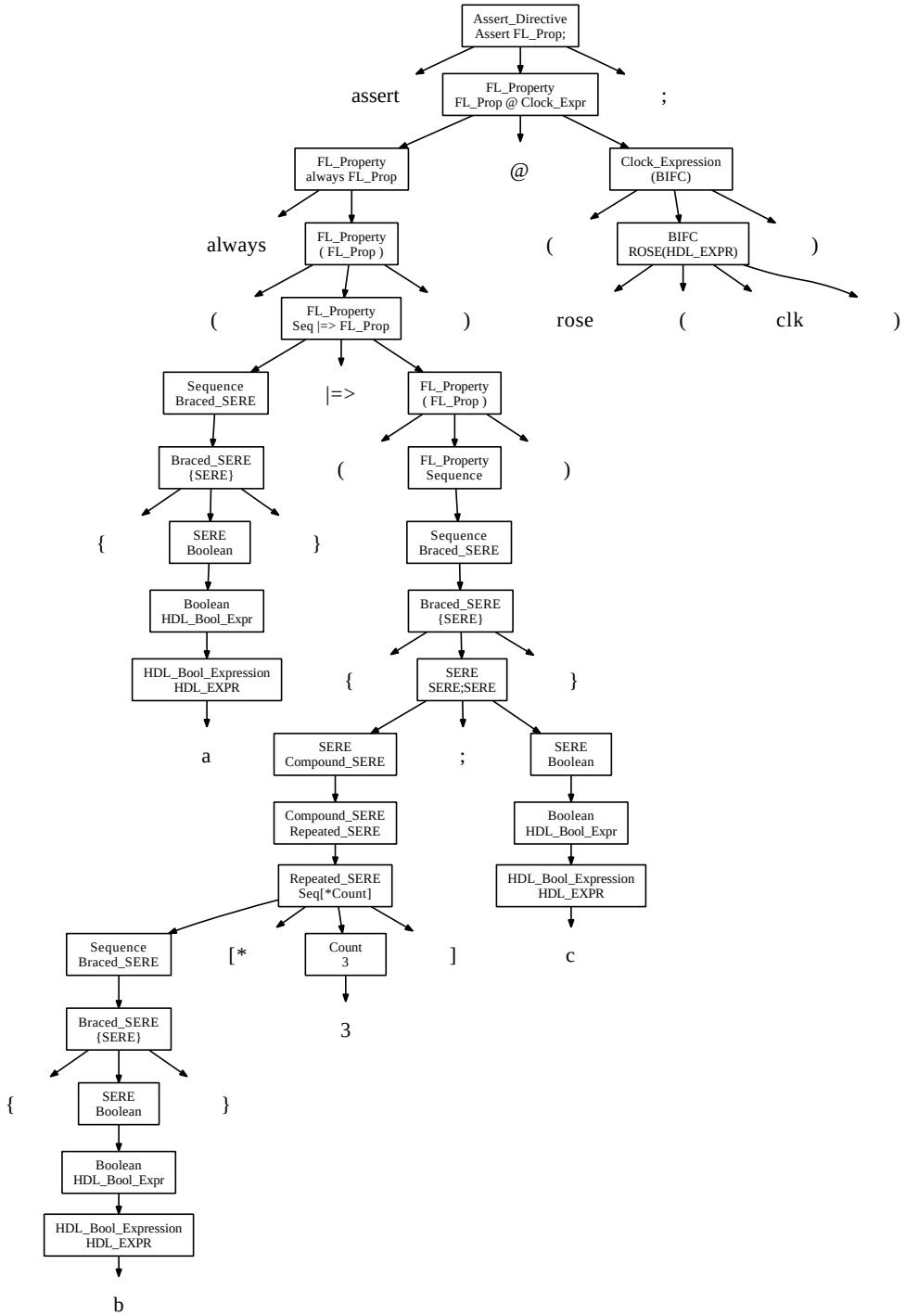


Figure 29: PSL parse tree example, generated automatically by psl2hdl.

Once a PSL formula is parsed, we apply Simple Subset checks (see Table 6) to the parse tree, to verify that the PSL formula meets the constraints of the Simple Subset. Next, we apply the rewrite rules (see Tables 7 and 9), so that the PSL formulas include only the "base case" elements, while maintaining the result in a grammatically valid PSL parse tree.

When parsing native boolean expressions in the underlying HDL (primarily Verilog), we preserve their structure in the syntax tree. Native HDL identifiers can be any alphanumeric. The boolean expressions can include parentheses, the operators AND, OR, NOT, NAND, NOR, XOR, XNOR, and $\leftrightarrow$. During parsing, the boolean operators are all unary or binary, and hence no boolean expression node in the tree has more than three children. In order to have more computational flexibility, after the rewrites we change from this format for boolean expressions, which complies with the grammar rules, to a list-based prefix-notation format. For example, consider a boolean expression in binary-tree format: (a AND (b AND (c AND d))). The expression is more succinctly represented in equivalent prefix-list notation: [AND a b c d]. In addition to this conversion, we reduce non-primitive boolean expressions (e.g., XNOR, $\leftrightarrow$, etc.) to forms using only the basic operators OR, AND, and NOT, using a function called *expand_nonprimitives*. We maintain the boolean expressions internally in disjunctive normal form (DNF), and implement a variety of simplification algorithms and boolean utilities.

In psl2hdl, automata are represented in the usual way, modeling the states, start state, accepting states, transition table, and the input symbols, but we also add a *mode* field to indicate whether the automaton is in *conditional-mode* or *fail-mode*. The mode field tells us how to interpret the output of the checker. The automata are generated and combined in accordance with the algorithms described earlier in this chapter. NFA-to-DFA conversion (determinization) and DFA minimization are also implemented as previously described.

In order to facilitate the recursive automata composition, we modified the *psl_tree* construct, so that each node in a *psl_tree* can have an automaton "attached" to it. We call this combined structure a *psl_tree_hybrid*. The reason this is necessary is because, at each node in the *psl_tree*, the automaton for that node derives from the automata of its child nodes, combined in some way based on the operator the node represents (e.g., sequence concatenation). We implemented a function called *generate_root_automaton*, which takes as input a *psl_tree_hybrid*, whose PSL parse tree elements are filled but whose automata are undefined. The function begins by creating base automata at the leaf nodes, and works its way up the tree (using a modified depth-first-visit algorithm), composing the child automata at each parent node in the tree, in accordance with the combination rules discussed in this chapter. At the end, the *psl_tree_hybrid* node at the root of the parse tree has attached to it an automaton for accepting the input sequences associated with that PSL formula.

Once the automaton is final, we convert it to its HDL representation, using a function called *convert_automaton_to_checker*, which also takes "Verilog" or "VHDL" as an argument, to specify either output format. The converter determines the HDL file's inputs, outputs, and internal signals, then creates all the necessary signal assignments to model the automaton. It outputs a text design file that can be immediately copied into an overall hardware design and connected to the modules to be checked.

Though we do not have access to the code used by Boulé and Zilic in their MBAC checker-generator, they do describe its capabilities [6]. We can also compare psl2hdl with Findenig's SynPSL tool [4]. In

addition, we can compare all the checker-generator tools with the native assertion support supplied in the latest commercial verification product from Mentor Graphics, called QuestaSim [81]. See Table 13 for a summary.

Feature	QuestaSim	SynPSL	MBAC	psl2hdl
Parse All of PSL Simple Subset	✓	✓	✓	✓
Implement All Simple Subset Assertions	✓		✓	✓
Create Synthesizable Checkers		✓	✓	✓
Implement Ranged SEREs		✓	✓	✓
Generate Abstract Syntax Tree Output				✓
Parse All of Underlying Flavor (Verilog)	✓		✓	
Support Built-in Function Calls	some		some	some
DFA Minimization			partial	✓
Full Boolean-Layer Optimization				✓
VHDL and Verilog Output				✓

Table 13: Comparison of PSL assertion support tools.

E. METHOD COMPARISON

In this section, we contrast our own checker-generator implementation with the method of Boulé and Zilic, on which it is based.

Boulé and Zilic developed an automata-based checker-generator method, called MBAC, for converting PSL formulas into synthesizable HDL constructs [71]. MBAC is most comparable to a predecessor called Formal Checkers (FoCs), a commercial system developed by IBM researchers [5], but Boulé and Zilic showed that MBAC is more efficient in many cases, and handles PSL functions that FoCs does not [6]. MBAC is also the first published PSL checker-generator method to use an entirely automata-based construction, for both SEREs and Properties, with all the algorithms described in detail [6].

For this research, we implement the checker-generator method as described by Boulé and Zilic, except for the minor exceptions noted in this chapter. *Our principal departure from their method is in the application*. We show how PSL formulas specifying prohibited behaviors can be deduced from a processor's architectural specification, then added to a processor implementation as checkers, in order to dynamically detect malicious inclusions that express themselves in a way that violates one or more of the prohibited behaviors. To our knowledge, the method presented here is the first application to:

- Specify prohibited behaviors for a processor implementation using PSL, based on language in the processor's architectural reference documents.
- Use a PSL-based checker-generator for detecting violations of the specified prohibited behaviors at runtime.

Though it was not our primary goal, we provided several enhancements to the Boulé and Zilic checker-generator method, including:

- Implementation of full boolean-layer simplifications.
- A structured argument for the full method's soundness and completeness with respect to the PSL formal semantics (see Chapter IX).
- Support for both VHDL and Verilog output.
- Full DFA minimization.
- PSL abstract syntax tree output.

We have also placed the code for our tool in the public domain. We employed a PSL lexer-parser provided by Findenig [4] under a Creative Commons license, but the other code for our tool was originally created for this research. We are not aware of any other open-source or public-domain tool for automata-based checker-generation of all PSL Simple Subset formulas, including all the features described in this chapter.

F. APPLICATIONS

The importance of being able to create synthesizable assertion-checkers for hardware is that it allows us to verify dynamically, in physical systems, whether our assertions hold. However, when using assertions to characterize and detect permitted and prohibited behaviors, much of the focus will still be on simulation. Modern design tools, like Mentor's QuestaSim, have recently added support for some assertion-checking in simulation. In addition to PSL, SystemVerilog Assertions (SVA) are popular, for example.

The basic workflow for checking security assertions, which identify prohibited behaviors, is illustrated in Figure 30. The figure describes the process outlined in this chapter. Starting with a processor architecture and a particular implementation of it, we map the prohibited behaviors into PSL assertions, using (from the implementation) the named circuits which carry out the functions described. From there, the PSL assertions are parsed, and the rewrite rules applied. From the PSL base cases, we construct checker automata, which are in turn converted into synthesizable HDL modules. The checker modules are copied into the design and connected to the appropriate input and output signals. With the checkers in place, we can run a simulation testbench, or proceed with FPGA synthesis or floorplanning for fabricating silicon.

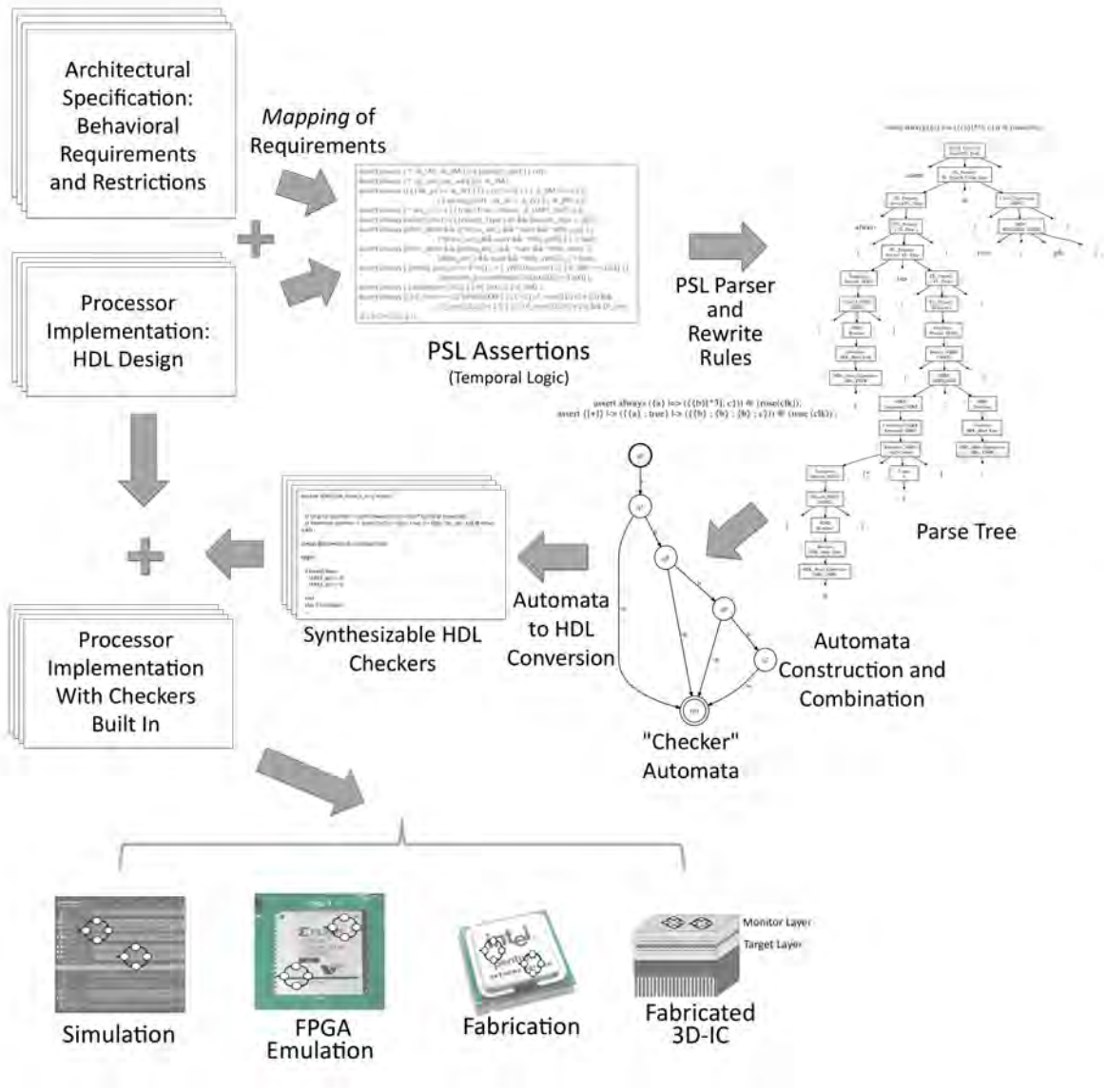


Figure 30: Workflow for synthesizable "security checkers."

1. Simulation

a. Basic Simulation

Once a set of assertions has been added to a processor design, we need to run the design through behavioral simulation, using any of a number of commercial and open-source products. Functional assertions will fail if there are implementation errors (bugs), while assertions implementing security requirements will fail if they observe prohibited behaviors.

In software simulation, it generally does not matter much if we evaluate our assertions under simulation using a software tool's built-in assertion-evaluation capability, or monitor the *hold* and *fail* signals of our synthesizable checker modules; if the latter are implemented correctly, they should agree. We do find it useful, during this stage, to compare the two outputs ("soft" assertions and checker hardware modules) as a cross-check, to gain confidence that the synthesizable checkers are semantically correct. In the rare case that an assertion cannot be written in a Simple Subset format, it may be possible in the future to implement it using commercial simulation tools, but as of this writing, leading products like QuestaSim still only implement most of the Simple Subset [81].

b. Adding Coverage

One of the most difficult issues faced by hardware engineers during verification is generating a testbench which exercises the design as thoroughly as possible. According to Boulé and Zilic [6]:

> In dynamic verification with assertions, proper care should be given to build a testbench that covers, as [completely] as possible, a meaningful and relevant set of scenarios. If an assertion did not fail because of 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 (emphasis added).

Coverage comes in many forms, such as code coverage, finite-state-machine coverage, branch coverage, expression coverage, and several others. Our primary interest is in code coverage. Perhaps the most commonly used form, code coverage examines the behavioral HDL code during simulation, and notes when the behavior associated with a particular HDL statement is executed. Software tools that implement code coverage can report, for a given simulation run, how many times a certain behavioral statement was executed, for example, or what percentage of all the statements was executed.

Some researchers have tried to leverage the assumption that malicious circuits added to a high-level design (i.e., malicious HDL code) will not be easily activated by a testbench, because they employ a rare-event triggers; some examples of these were given in Chapter III. In the design-analysis approaches taken by Hicks et al. [52], and by Banga and Hsiao [50], the investigators search for circuits which are either 1) activated only by statistically very unlikely input combinations, or 2) which fail to be activated by a testbench (as indicated by statement coverage of less than 100%). Recently, though, Sturton, Hicks, Wagner, and King showed how an adversary can avoid rarely-used/unused circuit identification (UCI) techniques, by "piggybacking" malicious signals on top of more frequently used circuits (occupying previously unused logic combinations), instead of using their own separate, dedicated circuits [2].

Fortunately, UCI/code-coverage techniques can be used *in concert with* assertion checkers, since they are complementary. The rationale behind the combined approach is that a well-designed testbench should exercise all, or nearly all, portions of a design; sections *not* exercised could be the result of 1) an incomplete testbench, 2) an extraneous piece of the design, or 3) rare-event-triggered malicious circuitry.[14] The general strategy is as follows:

[14] We implicitly assume, due to the nature of hardware designs, that signal values do not change *spontaneously*, but are only commanded to change due to some input stimulus through a logic gate. This is reminiscent of the observation of Datta et al., that it is obvious but necessary to assume that system permissions do not change on their own, to prove certain properties about the system [82].

- Design a thorough testbench, with the goal of achieving 100% coverage [44].
- Implement the checkers (synthesizable modules, modeling behavioral restrictions form the architecture).
- If we can reach 100% coverage (especially statement coverage) in all design units with the checkers active, then the design does not violate the security policy, as expressed by the checkers (though the design *could* still violate a policy requirement that's been omitted from the specification or incompletely or incorrectly expressed by the checkers).
- If we cannot achieve 100% coverage with the checkers active, then at least the "covered" portion of our design does not violate the security policy, as expressed by the checkers, and we can focus on the "uncovered" portions for further analysis (such as manual examination, or driving individual unit signals with the testbench to force greater unit coverage, for example).

By using this combination of techniques, we can detect MIs when they are activated by the testbench, or else constrain our search to a very small portion of the design when they are not. We illustrate this during our experiments (see Chapter VIII).

Consider the following hypothetical illustration. Suppose a Verilog design has a few lines of malicious code in it. Of course, in a real instance they would not be labeled with such obvious language, but rather obfuscated. Whenever the MI trigger is active, the circuits represented by this section of Verilog code will perform their function. Fortunately, these particular malicious behaviors are covered by some assertion-checkers. During a simulation run, executing these Verilog instructions corresponds to being "covered" during the run, such as by statement coverage (though we could also employ branch coverage and other forms). QuestaSim uses a checkmark (✓) to indicate that a statement has been covered, and an X to indicate that it has not.

In the first case, the simulation testbench has been run, but the testbench did not manage to trigger the malicious circuitry. The designer has not seen a checker violation, but notices that 100% statement coverage has *not* been achieved, and some ✕ marks remain (Figure 31).

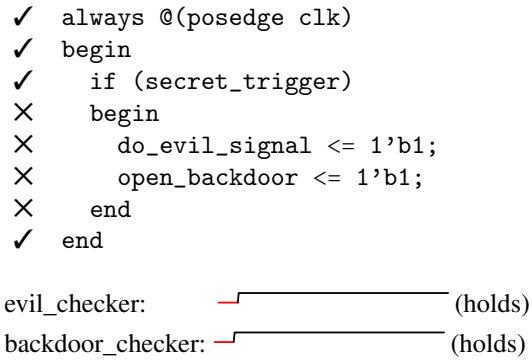


Figure 31: Incomplete code coverage example.

In the second case, the simulation testbench has been run (perhaps for longer, or with a better mix of random input vectors), and this time the testbench *did* trigger the malicious circuitry. The

designer has improved the statement coverage, possibly achieving 100%, but now the checkers fail (Figure 32).

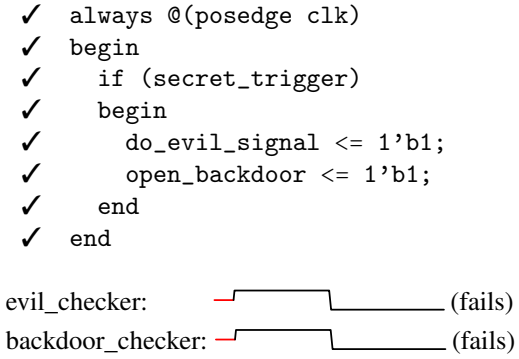


Figure 32: Complete code coverage example.

By combining checkers and statement coverage in this manner, we can increase our assurance that the design obeys the specified security requirements, and at the same time focus our search for malicious inclusions on smaller portions of the design, rather than analyze every line of design code.

2. FPGA Emulation and Fabricated Processors

In the evaluation of a processor design, FPGA emulation is orders of magnitude faster than ordinary simulation [6]. Therefore, it is useful to be able to perform assertion-based evaluation in FPGAs, a primary motivation behind checker-generator research. If we implement and synthesize assertion-based security checkers, along with assertion-based functional checkers, it will be possible to detect malicious functionality in FPGA emulation, as well.

The idea of having dynamic, runtime security checkers in fabricated silicon was suggested by Abramovici in 2009 [83]. According to his description, the security checkers are modeled as finite state machines which exist in reprogrammable portions of a processor, and monitor the activity of the processor by way of routed groups of signals. An advantage of this arrangement is that the "security policy" may be reprogrammed as desired. A disadvantage is the cost of manufacturing this type of chip. Also, there is no specified methodology for constructing the finite state machines; they must be provided by the customer.

Similarly, a customer procuring processors for a high-assurance application may desire to leave a set of PSL-based security checkers in a batch of fabricated ASICs. On detection of a prohibited behavior, the checker circuits could be programmed to activate a certain output signal, cause an interrupt, restart the processor, or signal the operating system in some way, depending on the application. This way, if an MI trigger condition was never introduced in simulation or emulation (due to the size and complexity of the design), but it did eventually occur in fielded operation, the malicious behavior might still be detected. Boulé and Zilic mention the potential utility of silicon-fabricated assertion checkers, in the context of functional verification [6]:

> Assertions can further play an important role in post-fabrication silicon debugging, where *assertion checkers are purposely left in the fabricated silicon*. 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 online status monitoring. In this way *a device can automatically assess its operating conditions, whereby the assertion checkers are used as a means of monitoring the device*. [emphasis added]

An advantage of leaving security assertion checkers in the design is that the same behavioral checks made in simulation and emulation may be carried forward to silicon fabrication, potentially detecting subversions made during manufacture. A disadvantage is the addition of area and power requirements, with potential impact to operating speed, because of the added circuits. We explore this overhead cost in Chapters VIII and IX .

3. 3D Processors

Another potential application of hardware-based, synthesizable assertion-checkers is in three dimensional integrated circuits (3D-ICs). Interest in 3D-ICs has increased significantly in the last decade because of the performance limits imposed by ever-decreasing feature sizes in traditional, single-layer chips. In a 3D-IC, two or more computational layers are fabricated separately, then joined using one of a wide variety of techniques.

There are two broad categories of 3D interconnect: coarse grained interconnect, which is already widely used, and fine-grained interconnect, which is not. In a coarse-grained 3D interconnect technique, such as wire bonding, the processing layers are large components, connected at their edges, using a relatively small number of inputs and outputs between the layers. In a fine-grained interconnect, processing layers are joined in their interiors by as many as thousands, or tens of thousands, of connections [84].

Coarse-grained interconnect technology, or "die-stacking," is in commercial use today. Though it provides fewer interconnects than fine-grained technology, it permits easy re-use of traditional 2D processing layers. Figure 33, from Lim [85], shows a cross section of the A4 processor, which uses die-stacking to locate two memory dies on top of a processor die; the physical proximity reduces latency between the processor and memory. In this cutaway image, the layers are visible but the interconnects are not. Because this type of coarse-grained application connects dies but does not fully connect the interior device and metallization layers, it is sometimes referred to informally as "2½-D" technology, as opposed to full "3D."

Figure 33: Apple A4 processor cross-section.

Fine-grained 3D interconnect technology is more challenging to implement, and is not yet in wide commercial use, though many companies are working to improve the fabrication tools and methods [86]. In a fine-grained 3D application, low-level components within a device on one layer can communicate directly

with their counterparts on another layer. The most common interconnect used is the Through-Silicon Via (TSV), but other types have been demonstrated [87]. A TSV is a metal wire, typically about 1μm in diameter, that runs from the metal layers of one die to the metal layers of an adjacent die. TSVs may carry ordinary data, but are also necessary for carrying power and ground signals, as well as distributing clock signals and dissipating heat. In 3D-ICs, the layers are manufactured individually, then joined together in a separate fabrication step.

There are many different fine-grained 3D fabrication processes being studied; here we illustrate, in Figure 34 (adapted from Loh [84]), two common types under development. In example (a), face-to-face bonding, the metal layers face each other and are joined at the bonding interface, using TSVs or some other interconnect. With face-to-face bonding, the interconnects do not traverse the device layers, minimizing die-area impact, but the method only works for two-layer designs. In example (b), face-to-back bonding, TSVs connect the metal layers of each die, but they must pass through the device layer of the upper die. In face-to-back bonding, the TSVs consume area in the device layers they pass through, but the method can be repeated to accommodate more than two layers. In either case, the interconnects must be precisely aligned when the layers are joined.

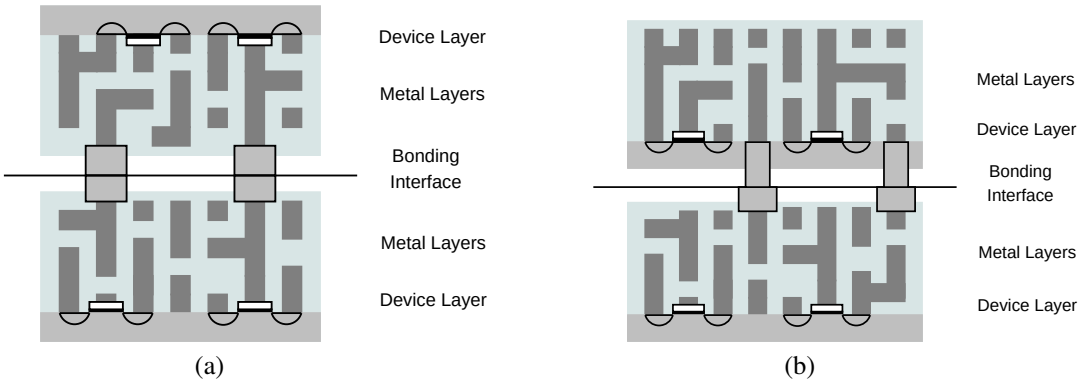


Figure 34: Face-to-face bonding (a), and face-to-back bonding (b).

Fine-grained 3D integration introduces several manufacturing challenges, and will require better design tools before it is widely used in practice [88]. However, it offers the possibility of greatly increased inter-layer bandwidth, compared to coarse-grained 3D, due to the greater number of interconnects [84].

a. 3D Monitoring and Security Applications

The advent of 3D-IC technology has naturally led to new application proposals. Mysore et al. introduced the idea [89] of "Introspective 3D Chips." In their method, a 3D "monitor" layer is attached to, and used to analyze, a "target" layer. The target layer would ship to consumers without the monitor layer attached; the monitor layer would primarily be used by developers for high-fidelity analysis of the target. The authors described how the monitor layer can be used for performance analysis, optimization, and bug detection, for example.

Huffmire et al. investigated the use of a 3D "control" layer for security applications [90]. They described three classes of 3D applications:

- *Passive Monitoring*: In these systems, the control layer observes the actions of the target layer, but does not interrupt its operation. Examples include: audit logging, information flow tracking, and runtime security and correctness checks.
- *Isolation and Protection*: Here, the control layer can override or reroute signals in the target layer, in order to enforce some isolation or access policy.
- *Secure Alternate Services*: The control layer can be used to provide, for example, trusted cryptographic services, to be accessed by the target layer.

For any of these methods, the potential advantage of implementing monitoring or security features by way of a separate layer is that a commodity target processor can be assembled with the extra features of the control layer only for those customers requiring it, and shipped without the additional layer for the general market. In fact, in the same spirit, some hardware modules that remain in commodity processors today, such as JTAG interfaces or other external debug modules, could eventually be *removed* from those commodity processors, freeing up some power and area for more primary computation circuits. A conceptual depiction is shown in Figure 35.

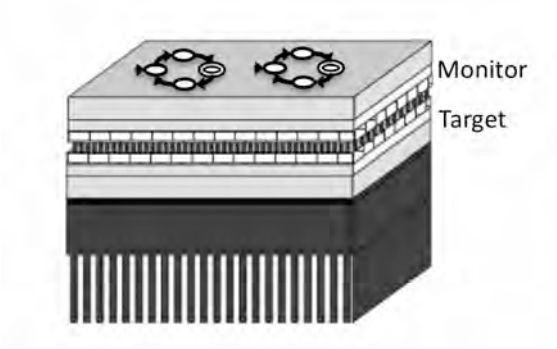


Figure 35: 3D-IC concept, showing target layer and monitor layer.

Bilzor presented the idea of a 3D Execution Monitor, or 3D-EM, in which a control layer observes the behavior of the target layer, and evaluates it against some behavioral requirement, as expressed using a state machine [91]. The method is similar to one proposed by Abramovici, for 2D-only systems [83]. Bilzor et al., described a general workflow for creating synthesizable security checkers, which could be used to implemented a 3D-EM [65]:

- From the architectural specification, identify any prohibited behaviors.
- From the processor design, identify the circuits which execute the related functions.
- In terms of the identified circuits, create PSL assertions that define observation of the prohibited behaviors.
- Using the available automated tools (like psl2hdl), translate the assertions into synthesizable HDL code.

- Attach the created HDL entities (security checkers) to the design, and use them to complement functional verification, typically in simulation or FPGA emulation.
- If desired, fabricate the security checkers in their own control layer, or 3D-EM. In the target processor design, add TSVs to export the monitored signals to the control layer.

b. Viability and Limitations of the 3D Method

Compared to 2D-only methods, 3D fabrication offers potential improvements not only in throughput, but also area and power. In their example, Mysore et al. modeled the addition of an analysis-engine chip to a Pentium 4 processor, by 1) adding the analysis engine to the target chip itself (in the same layer), and 2) adding the analysis engine chip as a separate 3D monitor layer, which could be absent in a consumer version of the chip. They showed significant improvement in power overhead (34% in one example) of the combined monitor-target system using the 3D strategy, compared to 2D. Meanwhile, the 3D-support mechanisms (primarily stubs for the TSVs) which are added to (and remain in) the target layer increase its power and area by only .9% and .021mm^2, respectively [89].

Using a separate 3D layer for security applications does have some limitations [88], including:

- 3D fabrication processes are relatively complex and expensive, and may have a lower yield .
- Fine-grained 3D-specific design tools are still maturing.
- Adjoining two processor layers requires great alignment precision.
- The layout, or floorplanning, of a fine-grained 3D processor design is more complex than floorplanning a traditional processor die.

G. PROPERTY TYPES

The fundamental unit of PSL is the *property*. For example, specific PSL properties may be *asserted* (required to hold), or *assumed* (assumed to hold), during verification. The theoretical foundations for what constitutes a *property*, in the computer science context, are set forth by Lamport [76], Rushby [92], and Alpern and Schneider [63], among others. In this section, we examine assertion checkers for PSL properties, in terms of properties in the general sense. In this section, we will use the term *PSL property* when describing a PSL formula specifically, and just the term *property* by itself, when referring to properties in the general (set-theoretical) sense.

An assertion checker is a form of Execution Monitor (EM), since it enforces a security policy, i.e., it is a predicate on executions. According to Schneider, "A set of [execution traces] is called a *property* if set membership is determined by each element alone and not by other members of the set," and "A security policy must be a property in order to have an enforcement mechanism in EM" [34]. Not all security policies express properties. Schneider mentions that an *information flow* policy [28] is a security policy that is not a property [34], for example. It is easy to see from the PSL formal semantics that PSL properties in the Foundation Language (FL) are properties, because *satisfaction* of the semantics—by a PSL FL property, over

a given execution trace—is defined by that execution trace alone, independent of any others. On the other hand, the formal semantics for PSL Optional Branching Extension (OBE) formulas are defined in terms of *sets* of execution traces, and therefore OBE formulas do not express properties.

A property may be further classified as a *safety property* or a *liveness property*, described next. Alpern and Schneider proved that all properties, in fact, from a topological point of view, are the intersection of safety and liveness properties [93].

1. Safety Properties

Informally, a safety property specifies that some identified "bad thing" does not happen. The occurrence of the "bad thing" is a violation of the property. More formally, according to Lamport, a property Γ is a safety property "if and only if Γ can be characterized using a set of finite executions that are prefixes of all executions excluded from Γ" [34]. According to Lamport [76] and Schneider [34], security policies satisfying the following three criteria are safety properties:

- They are *properties*.
- The sets of executions described are *prefix closed*.
- Any execution rejected by an enforcement mechanism must be rejected after a finite period.

Schneider adds, "Safety-critical systems are, for the most part, concerned with enforcing properties that are safety properties... so it is natural to expect an enforcement mechanism for safety properties to have application in this class of systems" [34].

The PSL specification defines a PSL safety property as a PSL property containing only weak, non-negated operators [62]. It is natural to ask whether a formula that is a safety property, according to the *PSL* definition, is also necessarily a safety property, according to the formal *theoretical* definition. Although we believe intuitively that this must be true, it has not formally been proved [94]. Completion of this proof would be useful future work.

Based on this theoretical outline, we expect the PSL assertions we use to generally express safety properties, if they are to enforce a security policy in a processor.

2. Liveness Properties, Availability Policies

Informally, a *liveness propert*y specifies that some identified "good thing" eventually happens. The failure of the "good thing" to eventually occur is a violation of the property. According to Alpern and Schneider, "L is a liveness property if any partial execution α can be extended by some execution β such that $\alpha\beta$ is in L" [93]. Some examples of a "good thing" that might be required to eventually occur include: granting of control over a bus after a request, fulfillment of a memory access request, completion of interrupt processing, etc.

The important distinction between properties that *are* safety properties, and properties that are *not*, is that the latter are not enforceable by an Execution Monitor, according to Schneider [34] and Rushby [92]. Says Schneider, "If the set of executions for a security policy P is not a safety property, then an enforcement mechanism from EM does not exist for P" [34].

However, this limitation can be avoided by placing a finite execution bound on when the "good thing" must happen; if it has not occurred by the end of the finite bound, the property fails. This may be referred to as "limited liveness." Technically, a finitely-bounded version of a liveness property is no longer a liveness property, and in fact may be enforceable [34].

This type of property enforces an *availability policy*, as described by Gligor [95]. An availability policy specifies that a principal (such as a user or process) cannot be denied access to a resource for more than a specified amount of time (number of execution steps). If we express an availability policy in an assertion-checker, then, we expect that the PSL property will describe a limited form of liveness, i.e., it will have this finite bound, in order to be enforceable.

H. FAILURE REPORTING

1. Failure Response

Although it is not the central focus of this research, we should consider what action a checker might take upon detecting a failure. Some possible actions include logging the failure, notifying the operating system or a user through a communication channel, or terminating execution. The appropriate action will depend primarily on the *application* (i.e., the type of system in which the processor is fielded) and the *setting* (i.e., simulation, FPGA, or fabricated processor).

In general, we expect a failure of one or more individual checkers to be a failure of the overall system of checkers. This is in accordance with Schneider's description of an Execution Monitor (EM), in which an overall security policy is the *conjunction* of a set of component policies [34]. This *composability* of EMs is also true of a system of assertion checkers.

In simulation, we can cause execution to terminate if a checker fails, but usually we expect that failures will simply be logged for further analysis after the simulation run.

FPGA emulation may be used on systems under development, or on fielded systems. If an FPGA emulation is being used during the verification phase, as part of developmental testing, we would again expect to simply log failures for further analysis. When an FPGA is in a fielded, operational system, on the other hand, we may choose to log the failures but may also communicate them to an external entity, or perhaps shut down the board or shift to a *failsafe* mode of operation. In one of our developmental tests, for example, we code a hardware driver for the FPGA board's LCD, to indicate to the operator whether any checkers have reported failures (see Figure 36).

In a fabricated processor, the designer will need to decide whether a checker failure should result in the termination of execution or not, and if so by what means. For example, if the processor is serving a classified network communication node, it may be desirable to immediately shut down the node when a checker fails, to mitigate the risk of data compromise. On the other hand, if a processor is part of an airplane's flight control system, some type of graceful degradation may be appropriate, or perhaps no action at all, depending on what failure is reported. An operational processor that detects a failure may be able to employ a fallback mechanism to some minimal trusted configuration, such as the system proposed by McCune et al. [96], for example.

Figure 36: Example of FPGA-based checker failure reporting during our developmental testing, using the Plasma processor model from OpenCores.

2. Timing of Failure Reports

A limitation of PSL's formal semantics is that they do not specify when and how to report failures during dynamic evaluation [62]. As a result, for example, commercial products that support software evaluation of assertions may report assertion failures at different times, without being incorrect, according to the PSL specification. According to Eisner, "PSL defines whether or not a property holds on a trace—that is all. It says nothing about when a tool... should report on results of the analysis" [55]. As a result, we find it useful to follow the *first-fail* method of Boulé and Zilic, in which a failure is reported immediately, as soon as it is discovered [6]; this approach seems best suited for dynamic evaluation. For example, in a fail-mode automaton, at the first instance the *fail* state is visited, the checker reports a failure immediately, and the entire input sequence fails no matter what other input is seen after the initial failure.

I. SUMMARY

In this chapter, we discuss the creation of PSL-based security assertions from the text of an architectural specification. We show how the method outlined by Boulé and Zilic can be used to translate these assertions into synthesizable checkers, which can be added to a hardware design. We then propose methods for employing these security checkers in simulation (along with the use of code coverage), as well as FPGA emulation, and in traditional and 3D fabricated processors. Finally, we describe the types of properties we expect to employ. In Chapter VIII, we give an example of how the method can be used to detect the presence of malicious inclusions in a general-purpose processor design.

THIS PAGE INTENTIONALLY LEFT BLANK

VIII. EXPERIMENTAL DEMONSTRATION

A. EXPERIMENT PLAN

The idea of the experiment is to illustrate how processor malicious inclusions (MIs) can be detected at runtime using the method we have outlined. Because we design both the checkers and the MIs, the demonstration is academic in that regard. We are not aware of any *adversarial* experiments in MI detection using general-purpose processors, where one group designs the MIs, and another group attempts to detect them, but we believe this is important future work.

This experimental demonstration is a proof of concept, showing how the behavior of some MIs, representative of those observed to date, can be detected using assertion checkers that are based on behavioral requirements from an architectural specification. The experiment is novel in this regard; we are not aware of any application of assertion-checkers to detect MIs in a general-purpose processor design. Because the experiment is a proof of concept for a novel method, rather than a quantitative comparison between methods, the total number of MIs and checkers is not of central importance, and adding more of each would would not add value to this demonstration, in our estimation.

First, we take an open-source general-purpose processor model, whose design code and supporting tools are freely available, and create several "typical" MIs targeting it, following some of the examples in Chapter III. Next, we use the text of the processor's architectural specification to generate a set of representative security assertions, describing various prohibited behaviors, in PSL, knowing *a priori* that some of the prohibited behaviors will be expressed by the MIs. We convert the assertions into assertion checkers, using our tool, and install the checkers in the design. We verify the behavior of the synthesizable checkers against commercial software-based assertion-checkers, using the same PSL formulas. Finally, we modify the processor testbench and firmware to run, both with and without the MI triggers, and observe the checkers, verifying that they do detect the occurrence of the prohibited behaviors in question, when those behaviors occur.

Although we already know that some of the MIs we create will violate corresponding behavioral restrictions from the architecture, we use the experiment to determine the following:

- Can the behavioral restrictions be translated into PSL assertions?
- Can the assertions be converted successfully into checkers? Can the conversion be done efficiently, in terms of time and space required?
- Do the checkers correctly reflect the PSL semantics?
- Do the checkers correctly identify the offending behaviors, with no false positives and no false negatives?

The following sections describe the experiment in detail.

B. OPENRISC AND MINSOC INTRODUCTION

OpenRISC is an advanced open-source processor architecture, supported by many contributors, and hosted by OpenCores.org [97]. The current OpenRISC CPU design is the "or1200." The or1200 has its own full MIPS-style [66] 32-bit instruction set, with optional 64-bit, floating-point, and vector extensions. The or1200 and other processor designs in the OpenRISC architecture family are single-core and have a pipelined execution unit, exception-handling units, data and instruction caches, memory-management units, a Wishbone bus, a debug unit, and support for peripherals via the bus. A diagram of the or1200 architecture is in Figure 37 [97].

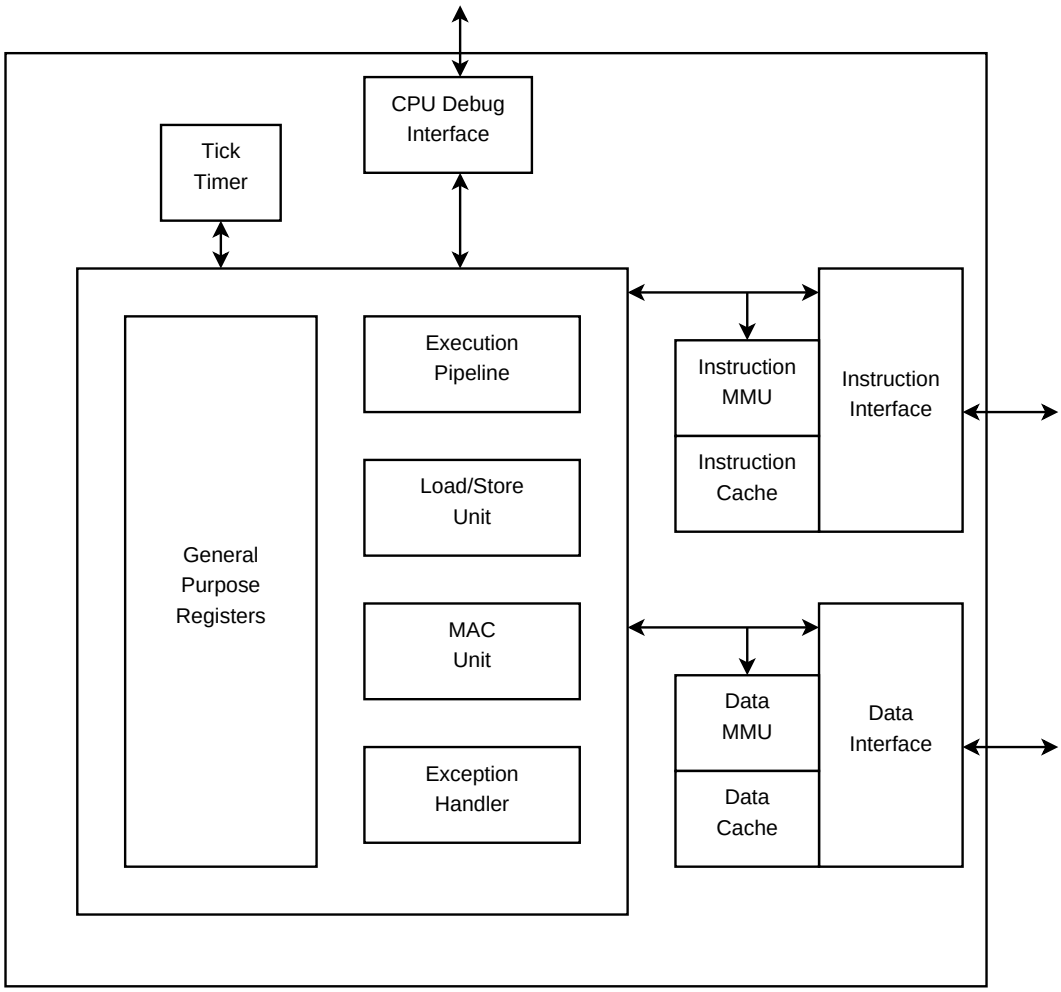


Figure 37: OpenRISC or1200 CPU processor architecture.

There are several different implementations based on the or1200; the implementation we used for these experiments is called MINSOC, for "minimal system on chip." MINSOC has an or1200 CPU plus on-chip memory, and includes Ethernet and UART units for input and output. An illustration of the MINSOC configuration is in Figure 38 [97].

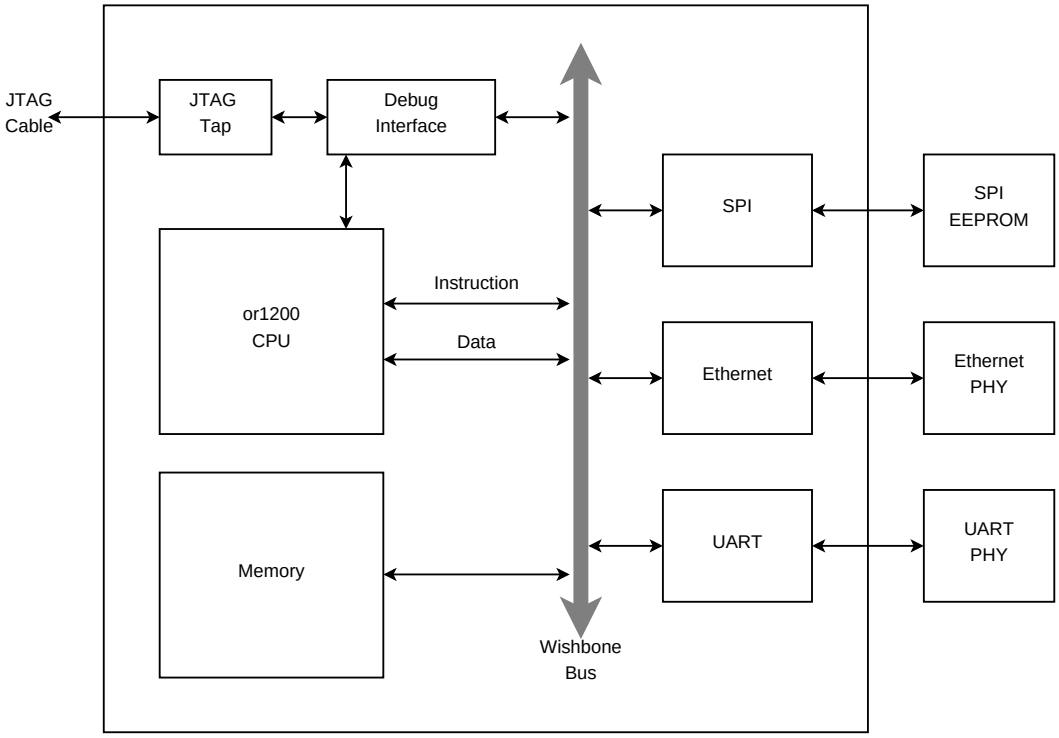


Figure 38: MINSOC system-on-chip configuration.

In order to exercise the MINSOC design, we use a standard testbench, supplied by the MINSOC creator, Fajardo [98]. The testbench provides Verilog functions for stimulating the external ports of the design with input data, modeling external physical connections.

Separate from the testbench, we use the OpenRISC "toolchain" to generate programs (binaries) targeted to run on the OpenRISC CPU; these programs are referred to as *firmware* in the OpenRISC documents. A firmware program is written in assembly-language format, or in C, then cross-compiled (from a Ubuntu [99] virtual machine platform) into an executable-linkable format, suitable for native execution on an OpenRISC platform CPU. In addition to the cross-compiler, the toolchain includes OpenRISC-targeted versions of standard GNU tools [100], such as *binutils*. Once the OpenRISC-targeted binary executable file is created, we copy it into the MINSOC design folders for simulation runs. We perform all simulations using Mentor Graphics' QuestaSim [81].

In the case of MINSOC, the firmware program can be loaded from external memory after CPU startup, or the firmware can be pre-loaded into on-chip memory at simulation start; we use the latter method, which reduces simulation times. For MINSOC, Fajardo provides a bootloader, written in assembly, which initializes the memory space and interrupt vectors, then hands control to a user-defined C program [98]. We construct such a C program, for example, to supply the trigger opcodes for the first of the attacks, described next.

C. MALICIOUS INCLUSIONS

We designed three MIs for these experiments.

MI #1 allows a software process running in the processor's User Mode to escalate its privilege level to Supervisor Mode, and is similar to other demonstrated MIs [18]. Once a process is running in Supervisor Mode, any number of software subversions are possible, as shown in the combined hardware-software attacks of King et al. [18]. The MI on-trigger is the opcode/data combination `l.addi r7, r0, 0xABCD`, which translates into the 32-bit binary value `0x9CE0ABCD`. The off-trigger is `l.addi r8, r0, 0x1234`, or `0x9D001234`.

MI #2 is designed to be able to leak secret data from memory out the UART port. Upon receipt of the input trigger text "Get Data" plus a memory location, MI #2 copies a sequence of bytes from the specified memory location, bypassing the normal memory access mechanism, and sends the data back out the port. No software is involved in this MI, which could also have been implemented on an Ethernet controller, or other I/O device. MI #2 is similar to several demonstrated by Jin, Kupp, and Makris [17].

MI #3, when triggered, disables the processor by causing its the internal reset signal to be continuously asserted. Normally, when initiated internally, the system would assert the or1200 reset signal for some finite time, and then de-assert it; when it is de-asserted, a chain of events occurs that leads to the start of the normal fetch-decode-execute processing cycle. Instead, MI #3 raises the internal reset signal and keeps it asserted (high) indefinitely; the clock continues to cycle, but no useful processing occurs. This can be considered a liveness property violation, because the user would normally expect a continuation of processing to follow an internal reset. MI #3 is triggered when the Ethernet terminal observes the input text "Shutdown" being received in a packet. There may be many ways to effectively disable a processor, using any number of different circuits, of course; this is just one example. MI #3 is similar to some of those mentioned in the "kill switch" review by Adee [8].

As we describe in the following sections, MIs #1 and #2 violate behavioral restrictions that are stated or implied in the specification, and checkers were constructed for those restrictions. The MIs are therefore detected by the respective checkers. However, while MI #3 violates the implicit restriction that signal values do not change spontaneously, there is no stated requirement that de-assertion must follow reset, which is violated by MI #3, so according to the methodology there is no corresponding checker, and so the MI remains undetected. MI #3 illustrates the case of an implicit violation for which there is no checker.

D. ASSERTIONS

In order to infer the processor's security requirements, we reviewed the OpenRISC and MINSOC manuals for statements that dictated certain behaviors. Though the manuals, like the processor design itself, are evolving, we were able to find several. In some cases, the behavior of a certain signal or component may be only partially specified, i.e., it is required to behave in a certain way under certain conditions, but its behavior is otherwise not dictated. As mentioned in Chapter VII, from a security standpoint, we need all the permitted and prohibited behaviors to be fully specified. That is, a given behavior should be either permitted or prohibited, but not neither, both, or unspecified/unknown. In the case where there are certain specified prohibited behaviors, and all others are implicitly permitted, we called it the *blacklist* approach. In

this experiment, we took the blacklist approach, searching for behavioral requirements, i.e., a behavior must proceed in conformance with the description, else the behavior is prohibited. We assume that an observed sequence, or behavior, which does not violate any of the specified requirements is a *permitted* behavior, and a behavior which violates one or more of the specified conditions is a *prohibited* behavior.

For this experiment, there are three governing documents: the OpenRISC architectural specification (which defines the OpenRISC architecture generally, including the instruction set), the or1200 CPU architectural specification (for the specific or1200 model CPU, which is a member of the OpenRISC family), and the specification for the MINSOC system-on-chip (which employs an or1200, plus other components).

We implemented the ten assertions listed below. Some of the assertions are simple boolean conditions, while others may span several clock cycles. Note that there is a one-to-many relationship between a requirement and the assertions needed to implement it, because often many signals and units, across varying components in a design, may collectively carry out a given function. For these examples, we use only one or two assertions per requirement. In developing a complete set of security checkers for an entire design, it may be difficult to know when all desired assertions have been generated, without exhaustively evaluating each architectural requirement against every design unit in the implementation. It would be valuable future work to better quantify, and perhaps automate, portions of the assertion-writing process, which we performed manually.

The assertions follow.[15] In each case, the requirement is stated, followed by one or more architectural references on which it is based, and a PSL assertion that enforces the requirement. Any observed behavior that violates a behavioral requirement (and hence, violates the assertion implementing it) is a prohibited behavior. Together, the requirements form a security policy, since they are a predicate on executions [34], identifying permitted and prohibited behaviors.

- *Requirement 1*: Group 0 special registers may only be modified if the CPU is already in supervisor mode.
  - *Source*: OpenRISC Architecture p. 21-3, Table 4-2
  - *Source*: OpenRISC Architecture p. 20: "Supervisor-level special-purpose registers are accessible only in supervisor mode of operation."
  - *PSL Assertion 1*: assert always {¬spr_we ; spr_we} ↦ ‘A_SM;
- *Requirement 2*: Supervisor mode is only entered from User Mode on reset startup, or exception entry.
  - *Source*: OpenRISC Architecture p. 338, "The OpenRISC 1000 provides two execution modes: user and supervisor. Processes run in user mode and the operating system kernel runs in supervisor mode."
  - *Source*: OpenRISC Architecture p. 252, "Processing of exceptions begins with a rise to supervisor mode."
  - *Source*: Specification for or1200, p. 24, "The Reset signal, when asserted high, immediately resets all flip-flops inside or1200. When de-asserted, or1200 will start the reset exception."

[15] Actual signal names are in lowercase. Identifiers in caps, like ‘A_SM, are placeholders for longer signal names, defined by compiler directives. For example, ‘A_SM represents the Supervisor Mode signal.

- *PSL Assertion 2*: assert always {¬‘A_SM; ‘A_SM } ↦ (except_start || rst);

- *Requirement 3*: Exception handling is only entered if one of the Table 6-3 mechanisms is activated.
  - *Source*: OpenRISC Architecture, p. 254
  - *PSL Assertion 3*: assert always {(except_start ↦ ((except_type > 0) && (except_type < 16)))};

- *Requirement 4*: Custom instruction opcodes are permitted, but must be declared in the implementation; unspecified custom instructions are not allowed.
  - *Source*: OpenRISC Architecture ch. 5
  - *PSL Assertion 4*: assert always { ¬(((if_insn[31:26] > 10) && (if_insn[31:26] < 17)) || ((if_insn[31:26] < 28) && (if_insn[31:26] > 21))) || (if_insn === 32'hXXXXXXXX)}

- *Requirement 5*: Instructions in the exception handling area of memory must only be accessed during processing of an exception, unless the processor is in supervisor mode, or during a reset.
  - *Source*: OpenRISC Architecture Table 6-2, p. 253
  - *PSL Assertion 5*: assert always { {(ex_pc >= 32'h00001000) || rst !==0 || ‘A_SM !== 0 || except_start ; (ex_pc < 32'h00001000) ; ‘A_RFE} };

- *Requirements 6-7*: A page fault must be generated if the MMU detects an access control violation for reads and writes.
  - *Source*: OpenRISC Architecture Table 8-14 and p. 283, “After a virtual address is determined to be within a page covered by the valid page table entry, the access is validated by the memory protection mechanism. If this protection mechanism prohibits the access, a page fault exception is generated.”
  - *PSL Assertion 6*: assert always {(dtlb_done & ((¬dcpu_we i & ¬supv & ¬dtlb_ure) || (¬dcpu_we_i & supv & ¬dtlb_sre)))} ↦ fault;
  - *PSL Assertion 7*: assert always {(dtlb_done & ((¬dcpu_we i & ¬supv & ¬dtlb_uwe) || (¬dcpu_we_i & supv & ¬dtlb_swe)))} ↦ fault;

- *Requirement 8*: The UART output signals may only change if a write has been commanded from the CPU.
  - *Source*: described, though not fully specified, in the MINSOC Architecture Manual, p. 5-6. The two clock cycle delay represented by *true; true* in the sequence is an artifact of the implementation.
  - *PSL Assertion 8*: assert always {¬we_o} ↦ {true ; true; stable(‘A_UART_OUT)};

- *Requirement 9*: A data change to the Ethernet output at the terminal should only occur if a transmit has been commanded, or during Ethernet or CPU initialization.

- *Source*: described in the MINSOC Architecture manual
- *PSL Assertion 9*: assert always {(mtxd_pad_o !== 4'h0)} ↦ ((prev (TxData[0]) !== 1'bX) || (WillTransmit) || ('A_SM === 1'bX));

- *Requirement 10*: The Debug Unit's Value and Control registers are only accessible from Supervisor Mode.
  - *Source*: OpenRISC Architecture, p. 299-300, "The debug value and control registers are 32-bit special purpose supervisor-level registers accessible [only] from supervisor mode."
  - *PSL Assertion 10*: assert always { ¬stable(dvr_x) } ↦ (('A_SM) || (rst ===1'bX));

It may appear as though we are duplicating some of the implementation's own logic, but that is not the case. The *names* of the circuits, and to an extent their behavior, as described in the PSL assertions, are particular to the processor implementation. However, the behavioral requirements derive from the *architectural specification*, and the assertion-checkers are external to the units they monitor; they receive copies of the monitored signals, but do not affect the internal operation of the monitored units.

Note that, in some cases, subtle language ambiguities need to be addressed during the mapping from text to PSL. For example, in requirement 10, the OpenRISC Architecture says that the debug value and control registers "are accessible from Supervisor Mode." As stated, this does not preclude them from *also* being accessible from User Mode. The requirement only becomes unambiguous if we either change it to say "the debug value and control registers are accessible from Supervisor Mode *and* User Mode," or change it to say "the debug value and control registers are accessible *only* from Supervisor Mode." This is an example of an incompletely or ambiguously specified requirement, which needs to be addressed during the mapping process.

E. SIMULATION RESULTS

We used psl2hdl to synthesize the ten assertions above into Verilog assertion-checkers, then added the Verilog modules to the design. We ran the simulation testbench with and without the MI triggers. Without the MI triggers, the testbench operates normally, and no assertions report failures. With the MI triggers, all three MIs are activated. MI #1 elevates the processor from User Mode to Supervisor Mode directly, then returns it to User Mode. MI #2 reads the trigger's memory location, surreptitiously copies eight bytes from memory starting at that location, and sends the data out the UART. Finally, MI #3 disables the processor. [16]

Results summary: As expected, the assertion-checker for Assertion #2 (supervisor mode) reports a failure when MI #1 operates (privilege escalation), and the assertion-checker for Assertion #8 (UART output)

[16] Because it stops the processor, MI #3 can be considered a disabling attack. It could also be construed to violate a liveness property, if one were specified in the architecture, such as "after an internal reset, the CPU eventually must resume fetching new instructions." We do not expect to be able to enforce liveness properties in general, with security checkers or any other runtime enforcement mechanism [34]. However, it is possible that certain "limited-liveness" properties could be enforced for other types of restrictions. Limited liveness differs from regular liveness in that it is not open-ended. For example, a system architecture could require that "after a reset, the CPU must resume fetching new instructions within 100 clock cycles." The latter is not open-ended, resembles a safety property, and is more practical to enforce. However, the target architecture does not constrain the behavior of the processor reset signal in either manner [101], so according to the methodology, there is no corresponding checker.

reports a failure when MI #2 operates (data exfiltration). Otherwise, all assertions hold at all times; see Tables 14 and 15 for the full results.

Assertion Number	Soft Assertion	HDL Checker
1	Hold	Hold
2	Hold	Hold
3	Hold	Hold
4	Hold	Hold
5	Hold	Hold
6	Hold	Hold
7	Hold	Hold
8	Hold	Hold
9	Hold	Hold
10	Hold	Hold

Table 14: MINSOC testbench: Assertion status *without* MI triggers active.

Assertion Number	Soft Assertion	HDL Checker
1	Hold	Hold
2	***Fail***	***Fail***
3	Hold	Hold
4	Hold	Hold
5	Hold	Hold
6	Hold	Hold
7	Hold	Hold
8	***Fail***	***Fail***
9	Hold	Hold
10	Hold	Hold

Table 15: MINSOC testbench: Assertion status *with* MI triggers active.

In order to obtain a cross-check of the checker semantics, we implemented the assertions in the simulation in two ways. First, we install a synthesizable checker, created by our tool, to model each assertion. Second, we implement each assertion natively in QuestaSim; we call this version a "soft assertion." The soft assertions are not synthesizable, but they allow us to cross-check the semantics of our checkers with the semantics implemented by the commercial simulator, QuestaSim. In this experiment, all of the software-based assertions and synthesizable assertion-checkers both indicated "hold" and "fail" at the same clock cycles, so the cross-check was successful, as indicated in Tables 14 and 15. See Figure 39 for an example. In the figure, the soft assertions are anchored by purple triangles, and the checkers are anchored by blue diamonds. In the simulation, the first assertion and the first checker both indicate a failure at the same clock cycle, by a red triangle and by a hold signal dipping to "0," respectively.

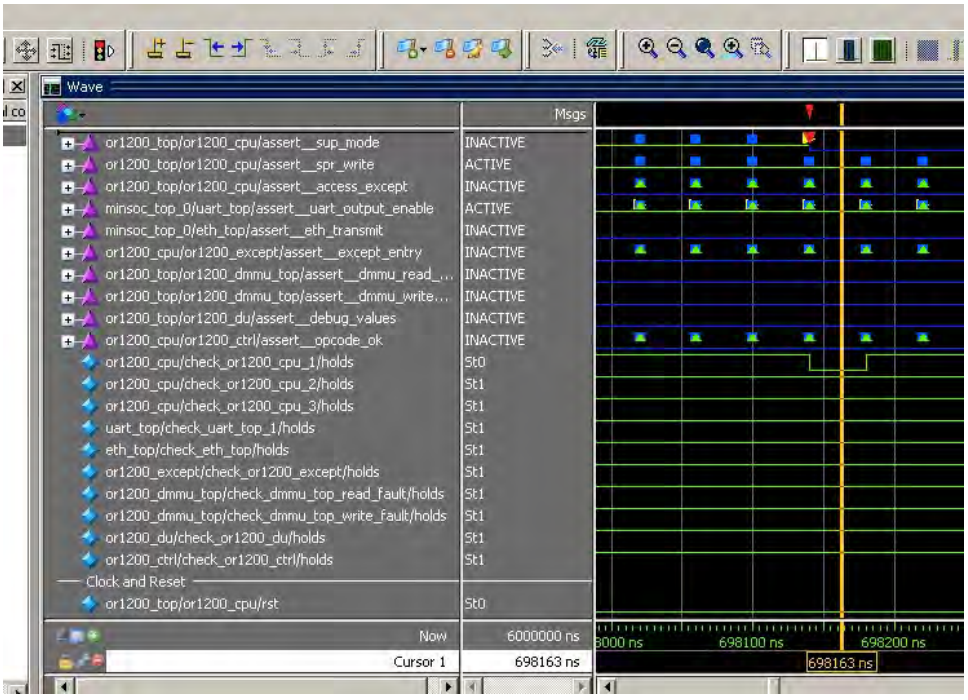


Figure 39: Cross-check between soft assertions and checkers.

We observed earlier that we would expect any MIs present in a design to be triggered if all the behavioral logic had been *covered*, as in 100% statement coverage, for example. We also observed that failure to trigger MIs in a design might be noted by less than 100% coverage being achieved, since the malicious circuits do not get activated (covered) during the simulation run. To illustrate this point, Table 16 shows the decrease in statement coverage, in some of the "infected" design units, when the MI triggers are turned off. We infer that, the larger the fraction of the design unit taken up by the MI, the larger the *decrease* in statement coverage, when the MI is inactive.

Unit	MI Trigger Active	MI Trigger Inactive
uart_top.v	100%	50%
eth_rxethmac.v	95%	73%
or1200_if.v	100%	95%

Table 16: MINSOC testbench: coverage in selected units, with and without an active MI trigger.

By combining security checkers and statement coverage in this manner, we can increase our assurance that the design obeys the specified security requirements, and at the same time focus our search for malicious inclusions on smaller portions of the design, rather than analyze every line of code.

F. EXPERIMENTAL OVERHEAD

To get a rough idea of the performance impact of adding the checkers, we synthesized the MINSOC design using the Xilinx ISE for a Virtex-5 target FPGA. With just these ten assertion checkers added, the

maximum clock speed was not affected (79 MHz either way), and the number of logic gates required increased .03%, compared to the MINSOC synthesized without checkers. All the resulting automata needed to implement the MINSOC assertions had five states or fewer. We analyze the overhead of the method in general, for a full processor, in Chapter IX.

IX. ANALYSIS

In this chapter, we analyze the method's soundness and completeness. We also estimate the overhead involved with application of the method in physical systems, and analyze the method's algorithmic complexity. We conclude with an assessment of the method's strengths and limitations.

A. SOUNDNESS AND COMPLETENESS

We begin with an informal discussion of the method overall, then examine the soundness and completeness of the checker-generator, with respect to PSL formulas and the semantics of PSL.

1. Cases Not Covered by the Method

The stated problem is to find all the MIs in a processor design. A complete solution would find all of them. We defined a malicious inclusion as "an unauthorized, undocumented modification to a piece of hardware or hardware design unit, which circumvents or subverts the hardware's specified security functionality or otherwise violates its documented behavioral restrictions." Based on the method outlined in Chapter VII, we analyze several cases where MIs might *not* be detected.

- In the following cases, it might not be possible to implement the method fully, due to the absence or incompleteness of the requirements:
  - A processor's architectural specification and other design documents do not state any behavioral restrictions.
    - In this case, the processor may be implemented in any manner that accomplishes the computational functionality, and no MIs will be detected by the method. The designer, or an attacker, is free to place backdoors or other subversions in the processor implementation, and they will not be detected by the method.
  - A processor's architectural specification and other design documents do state some behavioral restrictions, but the stated restrictions are ambiguous, contradictory, incomplete, or in other ways not able to preclude some subversions.
    - In order for a security policy to be enforced, it should be clearly and correctly stated. It should not be ambiguous or contradictory. We mention as possible future work the idea of analyzing a set of PSL formulas for inconsistencies and contradictions, which would be useful in this case. If MIs are to be detected as violations of specified behavioral restrictions, then those restrictions must be clearly stated.
- In the following cases, due to limitations of the existing checker-generator techniques, some expressible requirements may not be enforceable using our method:

- Eisner and Fisman claim they have not encountered a needed PSL restriction for a real hardware design that could not be formulated within the bounds of the Simple Subset [55]. However, it is possible that some hardware behavioral restrictions might not be expressible in the Simple Subset.
  - There exist some PSL formulas in the Foundation Language that are not in the Simple Subset [56]. Such a formula, if required to model a behavioral restriction, would not have a checker-generator procedure, in our method.
- Some portions of a processor, like memory storage units, are not well suited for application of our method:
  - The principal example of this is memory storage circuits. Though storage circuits are implemented using signals and gates at the fundamental level, it is not practical, for example, to have a dedicated checker monitoring every single bit or byte in a cache or other memory unit for correct operation (though it may be practical on small storage units, like a register file [102]). Rather, if we are concerned that the contents of a large memory unit may be subject to some malicious modification (in between the storage and subsequent retrieval of the data), it may be better to use some type of encryption, checksum, error-correction, or hashing scheme to detect the unauthorized modification, rather than an assertion-checker. The integrity of the data in hardware memory storage mechanisms against MIs is outside the scope of this research, but is an important topic for further study.

- In the following case, a requirement is expressible and enforceable, but our method would not detect all MIs, due to incomplete enforcement:
  - Suppose a requirement is properly stated in an architectural document, and converted into a PSL formula using the mapping process. Suppose the engineer creates a checker for signals x, y, and z in module A, but neglects to notice that signals x, y, and z are also present in module B, and does not install a checker for module B.
    - In this case, any MI targeting module B will not be detected, because the completeness of the method will often demand multiple checkers per requirement. It would be useful future work to automate the process of identifying and implementing these.

- In the following case, a requirement is expressible and enforceable, but the enforcement mechanism is subverted after it is added:
  - Suppose we have a clear, unambiguous, complete, and enforceable set of behavioral restrictions in the documentation. Suppose that we develop a complete set of checkers and insert them into the design, and perform a successful simulation, with no MIs noted, and leave synthesizable checkers in the fabricated final design.
    - If an adversary gained access to the processor design after insertion of the checkers and was savvy enough to subsequently insert an MI, he might also be able to disable one or more of the checkers, so that the MI is not detected at runtime.

2. Best-Effort Analysis

Under ideal conditions, this method will detect all MIs in a processor that have been inserted between the creation of the specification and completion of the high-level design. However, after the checkers have been implemented, subsequently inserted MIs will not be detected if the attacker is able to simultaneously disable all the checkers that would have detected the MI that was added.[17] The following set of conditions would constitute an "ideal," or "best effort," implementation of our method on a processor:

- A complete, consistent, unambiguous statement of all behavioral restrictions is included in the processor's architectural specifications.
    - Assessment of likelihood: very likely under ideal conditions only for small special-purpose hardware designs, but not very likely for a general-purpose processor design of modern complexity. Current-generation commercial processors are extremely large and complex, containing hundreds of component design units. The effort required to completely specify all behavioral restrictions for a processor would probably be cost-prohibitive in the general commercial case. However, the cost associated with this task on a scaled-down general-purpose processor model, designed specially for a set of high-assurance customers, may be acceptable for some customers.
- All behavioral restrictions are expressible in the PSL Simple Subset.
    - Assessment of likelihood: Likely. We use the comments from Eisner and Fisman's experiences as a reference; they claim they have never come across an assertion, for a real hardware design, that could not be expressed in the Simple Subset [55].
- A checker is generated for every module in the processor design that requires it (i.e., instantiation of checkers across multiple levels in a design hierarchy is complete).
    - Assessment of likelihood: Likely. Some portions of this process (creating multiple instances of a checker) could, in theory, be somewhat automated, which would be valuable future work.
- A complete testbench is developed, resulting in 100%, or very nearly 100%, coverage of the design. Any statements, branches, expressions, etc. not explicitly covered to 100% by the testbench are small enough in number to be analyzed manually for compliance with the stated restrictions.
    - Assessment of likelihood: Likely. Easier for smaller designs, more difficult for larger ones. In large-scale designs, though, such as a modern general-purpose processor, it's not uncommon to have multiple layers of testbenches, especially using a methodology like OVM [44], which facilitates a hierarchical, object-oriented approach to verification. Creating a good testbench is tedious, but engineers will generally strive to do so to support functional verification checks, in any case.

[17] Once a set of checkers is installed in a processor design, employment of various netlist obfuscation techniques, as well as anti-tamper methods, would be advantageous as a deterrent. These enhancements are outside the scope of this investigation.

In summary: though there may be some unexpressed behavioral requirements in an architecture, and there may be requirements that cannot be expressed in PSL, or in PSL's Simple Subset; under ideal conditions, assuming the high-assurance customer is willing to pay for the required extra analysis, and assuming either a scaled-down general-purpose processor not as complex as the current commercial state of the art, or a simpler special-purpose hardware design, the method is likely to successfully detect malicious inclusions whose action is characterizable as a violation of specified behavioral restrictions, if the checkers themselves are not subverted after installation.

3. PSL-to-Checker Soundness and Completeness

It is important to be sure the synthesized assertion-checkers behave correctly, i.e., that they properly hold or fail on a given sequence, according to the defined semantics of the PSL formula. Though our checker-generator method follows the recipe of Boulé and Zilic, they do not publish any formal justification of it with respect to PSL's formal semantics [6], so the following analysis is new.

In considering the overall correctness of the system, it is helpful to break the process into steps. We wish to verify that the semantics of the checker match the semantics of the original PSL formula on which it was based. The end-to-end soundness and completeness obligation can be stated as follows:

- "For any PSL formula in the Simple Subset, the method creates an assertion-checker whose semantics over input sequences are equivalent to those of the original PSL formula. In other words, for a given input sequence, the assertion-checker accepts an input sequence if and only if the input sequence satisfies the semantics of the PSL formula on which the checker was based."

The obligation can be stated as a structured argument in three parts, as described by the arrows in Figure 40. First, we must verify the correctness of a rewritten PSL formula, with respect to the original PSL formula (the rewrite process). Second, we need to verify the correctness of the generated automaton, with respect to the semantics of the rewritten formula (automaton generation process). Third, we have to verify the correctness of the HDL checker, with respect to the automaton from which it was generated (automaton-to-HDL conversion).

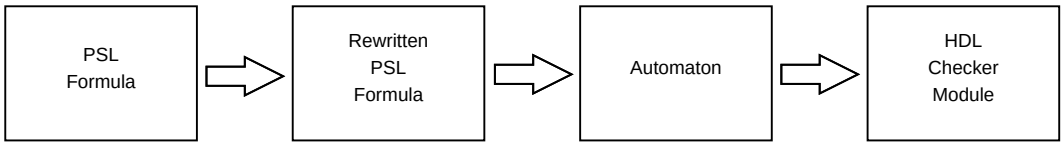


Figure 40: Conceptual depiction of the checker-generation process.

a. PSL Rewrite Rules

Part one was recently accomplished. A formal proof of the rewrite rules employed by Boulé and Zilic was developed by Morin-Allory, Boulé, Borrione, and Zilic [103]. The authors used the PVS theorem prover to prove the consistency of the rewrite rules, with respect to the formal PSL semantics. They uncovered minor inconsistencies in two of the rewrite rules with respect to PSL's Simple Subset definition,

and suggested modifications to the Simple Subset definition that overcome the problems identified.[18] The authors proved all the other rewrite rules correct with respect to PSL's semantics. For consistency, we adopt the slightly modified Morin-Allory PSL Simple Subset definitions [103], which do not affect the rewrite rules themselves.

The rewrite rules should not only be correct when applied, they should also ensure that any PSL Simple Subset formula will ultimately become, through successive application of the rewrite rules, a formula that fits the implementation base cases. To confirm this, we constructed a graph of the rewrite rule process (see Figure 41). Formats for all Simple Subset formulas that are not base cases are listed at the top of the unshaded nodes. A node represents a rewrite rule – for each, a PSL formula format is listed at the *top* of the node, and its rewritten version is listed at the *bottom* of the node. Each edge in the graph represents a resulting trigger to a subsequent rewrite rule or a base case; they can be thought of as forward dependencies. The shaded nodes at the bottom are the implementation base cases. We confirmed that there are no cycles in the graph and no orphan nodes, so that every (PSL Simple Subset) formula that is not already a base case is guaranteed to be *rewritable* into a base-case format. We are not aware of a comparable published analysis of the completeness of these PSL rewrite rules.

[18] None of the affected operators were used in the assertions employed in our experiments.

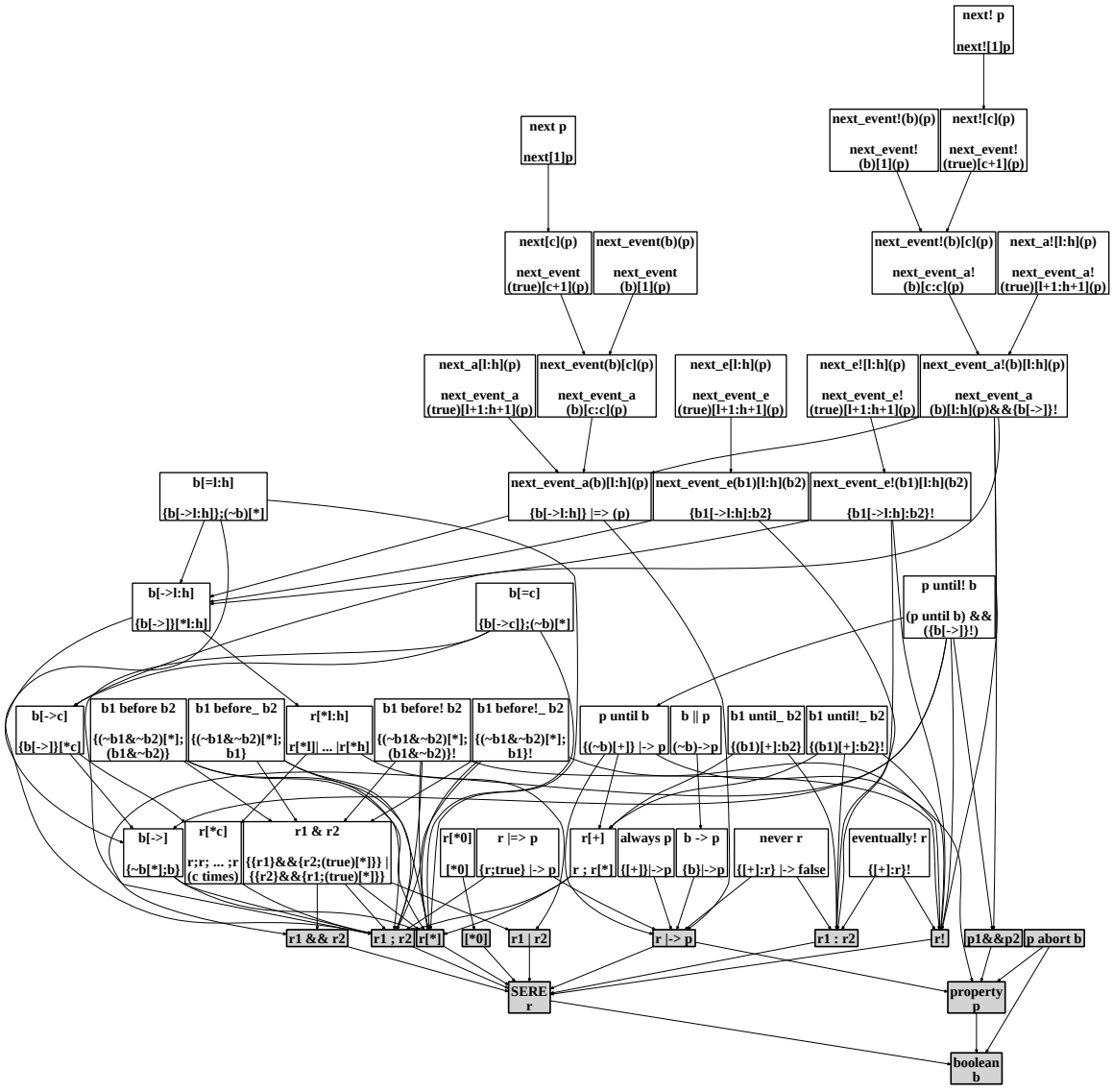


Figure 41: Rewrite rule dependencies. The base cases are represented by the shaded nodes, at the bottom.

b. Automata Implementation Base Cases

Part two of the argument requires that we show, with respect to the formal semantics of PSL, the correctness of the automata base-case implementations. We begin by excerpting the PSL formal semantics, over finite input sequences, from the specification [62], and then show that each of the base-case automata constructions is correct.

c. PSL Semantic Definitions

The semantics of PSL formulas are defined in the specification with respect to a model M, which represents the signals in a hardware system being evaluated [62]. The model M is defined there as a five-tuple (S, S_0, R, P, L), where:

- S is a finite set of states (s is a single state, $s \in S$)[19]
- $S_0 \subseteq S$ is a set of possible initial states ($s_0 \in S_0$)
- $R \subseteq S \times S$ is the transition relation (defining possible moves from a state s_n to a subsequent state s_{n+1})
- P is a nonempty set of atomic propositions.
- L is a "valuation" function, $L : S \longrightarrow 2^P$, which yields the set of assignments of the propositional variables for a given state. For example, if s_n represents the nth state, then the valuations of the propositional variables in that state are denoted l_n, which we may also refer to as the nth "letter" of the input word.

For this analysis, however, we are able to simplify the PSL model somewhat. First, note that the *states* referred to by S are states of the *hardware system* being modeled, rather than the states of an *automaton*. The definitions for S, S_0, and R are included to support multiple computation paths, as required by the OBE semantics. However, since we are not using OBE formulas, we note that the Foundation Language semantics, governing the Simple Subset, which we are using, do not use S, S_0, and R at all; the FL semantics rely only on the sequence of propositional input values (the input "word") [62], as defined by P and L. In addition, in our implementation, we use a single clock domain, evaluated over a *single computation path*, and require each signal to have only one value per clock cycle. Therefore, we argue that our simplification of the PSL *model* definition, below, into a PSL *dynamic system*, is sufficient for the semantic analysis that follows. For clarity, we call L an *input function* instead of a *valuation function*, since S, S_0, and R are omitted. We define a PSL dynamic system S as a tuple (P, L), where:

- P is a nonempty set of atomic *propositions*. Each proposition represents the value of a signal in the hardware model (as described in Chapter VI). At each clock cycle, each propositional variable has an input value of *true* or *false*. Each propositional variable is an alphanumeric.
- L is an *input function*, $L : n \longrightarrow 2^P$, which yields the set of assignments of the propositional variables for the clock cycle n, in a given "run" of the hardware being modeled. A run begins at clock cycle 0. The valuation of the propositional variables in clock cycle n is denoted l_n, which we also refer to as the nth "letter" of the input word. The initial values of the signals in the hardware system being modeled are represented by l_0.

This simplified version, which eliminates the need for the notion of *states* in the hardware system being modeled, avoids the potential confusion arising from having *states* in the hardware model and *states* in the automata model. The remaining semantic definitions, given next, follow the PSL specification once again, without simplification.

[19] States of the hardware system being evaluated.

Finite input words *u, v, w,* etc., are formed from sequences of letters, e.g., $v = (l_0 l_1 l_2 \ldots l_n)$. The *i*th letter of a word *v* is denoted v^i, the suffix of word *v* starting at v^i is denoted $v^{i\cdot\cdot}$, and the *i*th through *j*th letters of word *v* are denoted by $v^{i..j}$. The special symbols $\top$ ("top") and $\bot$ ("bottom") behave as follows: For every boolean expression *b*, $\top \models b$ and $\bot \not\models b$ (including $\top \models$ *false* and $\bot \not\models$ true). The *dual* of a word *v*, or $\bar{v}$, is obtained by replacing every assignment of $\top$ (top) with $\bot$ (bottom), and vice versa, in *v*. The notation $v^{0..j\top\omega}$ refers to an infinite extension of *v*, beyond the *j*th letter, with $\top$.

In the following sections, in keeping with the PSL specification, we use three different symbols that appear similar:

- $\models$ (boolean satisfaction)
- $\models\!\!\models$ (SERE satisfaction)
- $\vDash$ (property satisfaction)

Boolean expressions are formed as described in Chapter VII. As before, we denote as Φ the *logical evaluation function*, over the propositional variables in *P*, for boolean expressions *b*: $\Phi(b, l) \rightarrow \{true, false\}$. For a given boolean expression *b* and a letter *l*, if $\Phi(b, l) = true$, we say the letter *l* "satisfies" *b*, and write $l \models b$. For every boolean expression *b, true* $\models b$ and *false* $\not\models b$. Also, for a letter *l*, a propositional variable *p*, and a boolean expression *b*:

- $l \models p \Leftrightarrow p \in l$ (proposition *p* is *true* in *l*)
- $l \models \neg b \Leftrightarrow l \not\models b$
- $l \models true$
- $l \not\models false$
- $l \models b_1 \wedge b_2 \Leftrightarrow l \models b_1$ and $l \models b_2$

d. Automata Definitions, Restated

For convenience, we restate here the automata definitions from Chapter VII, since they will be referred to in the analysis that follows. Note that the simplified PSL system definitions now mesh with the automata definitions (*P* and *L* are unified).

- *P* is a nonempty, finite set of *propositional variables,* $p \in P$. At each clock cycle, each propositional variable has an input value of *true* or *false*. Each propositional variable is an alphanumeric.
- β is a finite set of *boolean expressions*, $b \in \beta$, formed in the usual way from the elements of *P*, plus the symbols for conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$), plus parentheses.
- *L* is the *input function* $L : n \longrightarrow 2^P$, representing the current assignments of *P* at the *n*th clock cycle. The set of input assignment values in clock cycle *n* is denoted l_n. We refer to l_n as the *n*th "letter" of an input word. The clock cycle *n* is a non-negative integer.

- Φ is a *logical evaluation function*, over an assignment l ("letter") of the propositional variables in P, for boolean expressions in β:
  $\Phi(b, l) \rightarrow \{true, false\}$. Φ evaluates propositional boolean formulas in the usual way. For example, if $b =$ "$x \wedge y$," and in clock cycle n, $l_n = \{x$=*false*, y=*true*$\}$, then we expect $\Phi(b, l_n) =$ *false*.

We define each propositional-logic automaton as a five-tuple: A = {Q, q_0, *L, F,* δ}, where:

- Q is a nonempty, finite set of states. Each individual state $q \in Q$ is described as either *active* (also representing true, or 1) or *inactive* (respectively false, or 0) during a clock cycle. The automaton initializes with all states *inactive* except the start state, q_0, which is *active*. Because the automata may be nondeterministic, it is permissible for more than one state to be *active* simultaneously.
- $q_0 \in Q$ is the *start state*.
- L is the *input function*, mentioned above, $L : n \longrightarrow 2^P$, which provides input values to the automaton, based on the current clock cycle.
- $F \subseteq Q$ is the set of accepting, or *final*, states. An input word is accepted by the automaton if and only if computation of the input word completes with one or more of the automaton's final states *active*. Note that acceptance will lag by one clock cycle; if an input word of length n is accepted, one or more final states will be *active* during clock cycle *n+1*.
- $\delta \subseteq Q \times \beta \times Q$ is the *transition relation* from state to state, via edges defined by boolean expressions. δ is a set of triples, { *(q,b,r)* | $q \in Q$, $b \in \beta$, $r \in Q$ }.

e. Implementation Cases

We now consider the base implementation cases one at a time. The automata may be either nondeterministic or deterministic, in general; however, fail-mode automata will always be deterministic by their construction. The semantic definitions listed come from the PSL specification [62].

- *Empty Set*. The PSL formula *false* accepts no input sequence, according to the definitions above. It is modeled by a single-state automaton (See Figure 18) with no transitions and no accepting states. Neither the PSL formula nor the automaton accepts any input sequence v; the language of both is Ø.
- *Empty Sequence*. By definition, the degenerate SERE "`[*0]`" accepts only the empty input sequence, ε. It is modeled by a single-state automaton with no transitions, whose start state is also an accepting state (see Figure 18). The language of both is $\{\varepsilon\}$.
- *Braced SERE*. The semantics for the braced SERE {r} are: $v \models \{r\} \Leftrightarrow v \models r$. In the construction, the automaton for {r} is identical to the automaton for r, so the semantic definition holds.
- *Boolean (Sequence)*.
  - Semantic definition: $v \models b \Leftrightarrow |v| = 1$ and $v^0 \models b$.
  - Automaton implementation: two-state conditional-mode automaton with a start state, final state, and a transition on b.

- If $|v| \neq 1$, then the automaton does not accept and v does not satisfy b. If $|v| = 1$, and $v^0 \models b$, then the automaton accepts after one clock cycle, and $v \models b$. The language represented by the SERE and the automaton are both simply $\{b\}$.

For the remaining cases, we employ the following format. First, the semantic definition from the PSL specification is stated. Next, the automaton implementation is reviewed. Finally, we analyze each direction:

- Assume an input word satisfies a formula, and show that the checker automaton holds on that input.
- Assume that a checker automaton holds on an input, and show that the input satisfies the formula.

After both cases are shown, we conclude that the input satisfies the formula (in accordance with the PSL formal semantics) *if and only if* the input holds on the checker automaton (in accordance with the construction given). A single automaton transition from state p to state q on an input letter l is denoted $\delta(p,l,q)$; multiple automaton transitions on an input word v, beginning at state p and ending at state q, are denoted $\hat{\delta}(p,v,q)$.

- *SERE Concatenation.*
  - Semantic definition: $v \models \mathrm{r}_1;\mathrm{r}_2 \Leftrightarrow \exists v_1, v_2$ such that $v = v_1 v_2$, $v_1 \models \mathrm{r}_1$, $v_2 \models \mathrm{r}_2$.
  - Automaton construction: given A_1 which models r_1, and A_2 which models r_2, construct A_3 by copying A_1 and A_2, then redirecting all edges inbound to final states in A_1 to the start state in A_2. Make all the final states in A_2 (but not those from A_1) into final states in A_3. Make the start state in A_1 the start state in A_3.
    - $Q_3 = (Q_1 - F_1) \cup Q_2$
    - $F_3 = F_2$
    - $q0_3 = q0_1$
    - $\delta_3 = \delta_1 \cup \delta_2$. Then, for all $d_{(p,b,q)} \in \delta_3 \mid q \in F_1$, change q to $q0_2$.
  - Analysis: Suppose automaton A_1 models SERE r_1, and automaton A_2 models SERE r_2. In other words, $v \models \mathrm{r}_1 \Leftrightarrow$ (there exists $\hat{\delta}(p,v,q) \mid p = q0_1, q \in F_1$) (respectively the same for r_2, A_2).
    - Suppose $v \models \mathrm{r}_1;\mathrm{r}_2$. Because $v_1 \models \mathrm{r}_1$, on input v_1, A_3 transitions from $q0_1$ to $q0_2$. Because $v_2 \models \mathrm{r}_2$, on input v_2, A_3 transitions from $q0_2$ to an accepting state in F_2 (hence, the same in F_3). Since $v = v_1 v_2$, $v \in L(A_3)$.
    - Suppose $v \in L(A_3)$. Subdivide v into a prefix v_1 and a suffix v_2, based on the point in the input sequence when reaching $q0_2$. Because the only edges from A_1 to A_2 are inbound, the computation in A_3 must pass through $q0_2$, and never re-enters the old A_1 states after leaving them, so the computation of suffix v_2 is determined entirely by the old states from A_2. A_3 can only accept v if $v_1 \in L(A_1)$ and $v_2 \in L(A_2)$. Hence, $v = v_1 v_2$, $v_1 \models \mathrm{r}_1$ and $v_2 \models \mathrm{r}_2$, so $v \models \mathrm{r}_1;\mathrm{r}_2$.
  - Conclusion: $v \models \mathrm{r}_1;\mathrm{r}_2 \Leftrightarrow v \in L(A_3)$.
- *SERE Fusion.*
  - Semantic definition: $v \models \mathrm{r}_1:\mathrm{r}_2 \Leftrightarrow \exists v_1, v_2, l$ such that $v = v_1 l v_2$, $v_1 l \models \mathrm{r}_1$, $l v_2 \models \mathrm{r}_2$.

- Automaton construction: Given automaton A_1 which models r_1 and automaton A_2 which models r_2, we fuse them by merging the incoming edges to the final states in A_1 with the outgoing edges of the start state in A_2. As a result, in the combined automaton A_3, each edge leading into a final state in A_1 now leads to a *successor* of the start state in A_2. The edges are combined by conjunction. If there are j edges coming into final states in A_1 and k edges leading out of the start state in A_2, there will be at most $j \times k$ matching edges in A_3. The construction eliminates from A_3 the final states in A_1 and the start state in A_2.
  - $Q_3 = (Q_1 - F_1) \cup (Q_2 - q0_2)$
  - $F_3 = F_2$
  - $q0_3 = q0_1$
  - $\delta_3 = \delta_1 \cup \delta_2$. Then, for all pairs of edges $d1_{(p1,b1,q1)} \in \delta_3 \mid q_1 \in F_1$ and $d2_{(p2,b2,q2)} \in \delta_3 \mid p_2 = q0_2$, delete edges $d1$ and $d2$ from δ_3 and add edge $d3 = (p1, b1 \wedge b2, q2)$ to δ_3.
- Analysis: Suppose automaton A_1 models SERE r_1, and automaton A_2 models SERE r_2. In other words, $v \models\!\models \mathrm{r}_1 \Leftrightarrow$ (there exists $\hat{\delta}(p, v, q) \mid p = q0_1, q \in F_1$) (respectively the same for r_2, A_2).
  - Suppose $v \models\!\models \mathrm{r}_1{:}\mathrm{r}_2$. By the semantics, $\exists v_1, v_2, l \mid v = v_1 l v_2,\ v_1 l \models\!\models \mathrm{r}_1,\ l v_2 \models\!\models \mathrm{r}_2$. By the construction of A_3 and the assumption that $v_1 l \models\!\models \mathrm{r}_1$, on input $v_1 l$, A_3 transitions to a state which follows the start state from A_2. From this state, A_3 will transition to a final state via input v_2. Hence, $v \in L(A_3)$.
  - Suppose $v \in L(A_3)$. Subdivide v into v_1, l, and v_2, such that l is the input causing a transition from the A_1 states to the A_2 states. As before, since the edge taken on input l is the only bridge between the states of A_1 and A_2, the v_1 portion of the input is computed by the A_1 states (and would have reached a final state on l), and the v_2 portion of the input is computed by the A_2 states (and reaches an A_2 final state). Therefore, $v = v_1 l v_2,\ v_1 l \models\!\models \mathrm{r}_1,\ l v_2 \models\!\models \mathrm{r}_2$, so $v \models\!\models \mathrm{r}_1{:}\mathrm{r}_2$.
- Conclusion: $v \models\!\models \mathrm{r}_1{:}\mathrm{r}_2 \Leftrightarrow v \in L(A_3)$.

- *SERE Closure*.
  - Semantic definition: $v \models\!\models \mathrm{r}[*] \Leftrightarrow (v \models\!\models [*0])$ or ($\exists v_1, v_2$ such that $v_1 \neq \varepsilon$, $v = v_1 v_2$, $v_1 \models\!\models \mathrm{r}$, and $v_2 \models\!\models \mathrm{r}[*]$).
  - Automaton construction: To represent the Kleene closure [75] of an input sequence accepted by an automaton A_1, we construct automaton A_2 as follows: reroute all edges inbound to final states to the start state instead. When the automaton accepts an input sequence, its computation path is simultaneously returned to the start state. In addition, since closure includes zero or more instances of a sequence, if the start state in A was not an accepting state, we make it into an accepting state in A_2, to accept the empty input sequence.
    - $Q_2 = Q_1$
    - $F_2 = F_1 \cup q0_1$
    - $q0_2 = q0_1$

- $\delta_2 = \delta_1$. Then, for all edges $d_{(p,b,q)} \in \delta_2 \mid q \in F_1$, change d to$(p, b, q0_2)$.

- Analysis: Suppose automaton A_1 models SERE r. In other words, $v \models \mathrm{r} \Leftrightarrow$ (there exists $\hat{\delta}(p, v, q) \mid p = q0_1, q \in F_1$).
  - Suppose $v \models \mathrm{r}[*]$. By the semantics, either $v \models [*0]$ or $\exists v_1, v_2 \mid v_1 \neq \varepsilon$, $v = v_1 v_2$, $v_1 \models \mathrm{r}$, and $v_2 \models \mathrm{r}[*]$. If $v \models [*0]$, then $v = \varepsilon$, and by the construction $v \in L(A_2)$, since the start state is an accepting state. If $\exists v_1, v_2 \mid v_1 \neq \varepsilon$, $v = v_1 v_2$, $v_1 \models \mathrm{r}$, and $v_2 \models \mathrm{r}[*]$, then we know v_1 would have been accepted by A_1, and is also accepted by A_2, which returns to its start state before computing the suffix v_2. Since $v_2 \models \mathrm{r}[*]$, the computation of v ends in a final state of A_2, so $v \in L(A_2)$.
  - Suppose $v \in L(A_2)$. If $v = \varepsilon$, then the first semantic condition is satisfied. If $v \neq \varepsilon$, then there must exist a partition of v into v_1, v_2 such that the prefix v_1 is computed on one cycle through A_2 (as if it were being computed by A_1) then returning to the start state, where v_2 will be computed in a manner semantically identical to v (i.e., beginning at the start state of A_2). Therefore, $v \models \mathrm{r}[*]$.
- Conclusion: $v \models \mathrm{r}[*] \Leftrightarrow v \in L(A_2)$.

- *SERE Disjunction.*
  - Semantic definition: $v \models \mathrm{r}_1 \mid \mathrm{r}_2 \Leftrightarrow v \models \mathrm{r}_1$ or $v \models \mathrm{r}_2$.
  - Automaton construction: Given input automata A_1 and A_2, we create a new automaton A_3 for sequence $\mathrm{r}_1 \mid \mathrm{r}_2$ by combining the start states of A_1 and A_2 into a single new start state for A_3, with all the other states and edges unchanged. All final states in A_1 and A_2 remain final states in A_3.
    - $Q_3 = Q_1 \cup (Q_2 - q0_2)$
    - $F_3 = F_1 \cup F_2$
    - $q0_3 = q0_1$
    - $\delta_3 = \delta_1 \cup \delta_2$. For all edges that were outbound from $q0_2$, make them outbound from $q0_1$ instead.
  - Analysis: Suppose A_1 models r_1 and A_2 models r_2. In other words, $v \models \mathrm{r}_1 \Leftrightarrow$ (there exists $\hat{\delta}(p, v, q) \mid p = q0_1, q \in F_1$) (respectively the same for r_2, A_2).
    - Suppose $v \models \mathrm{r}_1 \mid \mathrm{r}_2$. By the semantics, $v \models \mathrm{r}_1$ or $v \models \mathrm{r}_2$. Therefore, either $v \in L(A_1)$ or $v \in L(A_2)$. By the construction, $v \in L(A_3)$.
    - Suppose $v \in L(A_3)$. Since the start state, final states, and transitions come from A_1 and A_2, either $v \in L(A_1)$ or $v \in L(A_2)$, so $v \models \mathrm{r}_1$ or $v \models \mathrm{r}_2$, hence $v \models \mathrm{r}_1 \mid \mathrm{r}_2$.
  - Conclusion: $v \models \mathrm{r}_1 \mid \mathrm{r}_2 \Leftrightarrow v \in L(A_3)$.

- *SERE Conjunction.*
  - Semantic definition: $v \models \mathrm{r}_1 \;\&\&\; \mathrm{r}_2 \Leftrightarrow v \models \mathrm{r}_1$ and $v \models \mathrm{r}_2$.

- Automaton construction: As described in Chapter VII, given input automata A_1 and A_2 for SEREs r_1 and r_2, we create a new automaton A_3, which accepts the length-matching intersection, or conjunction, using a procedure similar to the classical product construction [75]. We generate *state pairs*, using one state at a time from each input automaton, and see what state each individual automaton (A_1 and A_2) would transition to on a given input. In the output automaton, A_3, each of these state pairs forms a single state. A state (state pair) is final in A_3 if both of its components were final in their respective original automata, A_1 and A_2. See Figure 23 in Chapter VII for an example.
- Analysis: Suppose A_1 models r_1 and A_2 models r_2. In other words, $v \models r_1 \Leftrightarrow$ (there exists $\hat{\delta}(p, v, q) \mid p = q0_1, q \in F_1$) (respectively the same for r_2).
  - Suppose $v \models r_1$ && r_2. By the semantics, $v \models r_1$ and $v \models r_2$. Therefore, $v \in L(A_1)$ and $v \in L(A_2)$, and so there exists a computation path for v in A_3 which leads to a state pair whose component states are final in A_1 and A_2, respectively, hence $v \in L(A_3)$.
  - Suppose $v \in L(A_3)$. By the construction, v has a computation path in both A_1 and A_2 leading to a final state, so $v \in L(A_1)$ and $v \in L(A_2)$, hence $v \models r_1$ and $v \models r_2$.
- Conclusion: $v \models r_1$ && $r_2 \Leftrightarrow v \in L(A_3)$.

- *Parenthesized Property*. In the semantics, a parenthesized property is handled the same as the non-parenthesized property: $v \vDash (p)$ iff $v \vDash p$. The automaton construction follows suit.

For the following semantic analysis, recall from Chapter VII that when interpreting both booleans and sequences as properties, we convert the automata to *failure mode*, using a *fail-mode automaton*, while a boolean or a sequence that is a SERE is interpreted in *conditional mode*, using a *conditional-mode automaton*. Informally, SEREs are detecting *occurrences*, while properties are detecting *failures to occur*. In our implementation, a property's checker holds if its automaton's *fail* state is *not* reached. See Figure 42, reprinted from Chapter VII, for the boolean example; the checker holds on an input sequence if the *fail* state of the automaton is *not* visited.

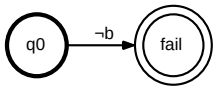


Figure 42: Boolean b, interpreted using property semantics.

- *Boolean (Property).*
  - Semantic definition: $v \vDash b \Leftrightarrow |v| = 0$ or $v^0 \Vdash b$.
  - Automaton construction: Given as input A_1 which models b as a SERE, convert it to a failure mode automaton A_2, as in Figure 42.
  - Analysis: An automaton A_1 that models b by SERE semantics will have two states (one start, one final), and a single edge on b, as in Figure 18.

- Suppose $v \models b$. By the semantics, $|v| = 0$ or $v^0 \Vdash b$. If $v = \varepsilon$, then A_2 does not reach the *fail* state, and the checker holds. If $v^0 \Vdash b$, then again A_2 is not able to reach the *fail* state (as in Figure 42), and the checker holds ($v \notin L(A_2)$, *fail* state not visited on input v).
- Suppose the checker holds ($v \notin L(A_2)$, *fail* state not visited on input v). It must be that b was true in v^0, or $v = \varepsilon$. We conclude by the semantics that $v \models b$.

- Conclusion: $v \models b \Leftrightarrow$ the checker holds on input v ($v \notin L(A_2)$, *fail* state not visited on input v).

- *Strong SERE (Property)*. Note, in the right-hand side of the semantic definition, the use of the SERE-semantics satisfaction operator ($\models\!\!\models$) for SERE r.

  - Semantic definition: $v \models \text{r!} \Leftrightarrow \exists j < |v|$ such that $v^{0..j} \models\!\!\models \text{r}$.
  - Automaton construction: Recall from Chapter VII that, given an automaton A_1 for SERE r, we convert it to a fail mode automaton A_2, then add transitions from each state, to the *fail* state, on detection of the end-of-execution signal.
  - Analysis: Suppose that conditional-mode automaton A_1 models r by SERE semantics. In other words, $v \models\!\!\models \text{r} \Leftrightarrow$ (there exists $\hat{\delta}(p, v, q) \mid p = q0_1, q \in F_1$).
    - Suppose $v \models \text{r!}$. Then $\exists j < |v|$ such that $v^{0..j} \models\!\!\models \text{r}$, meaning there is a prefix $v^{0..j}$ of v that SERE satisfies r. So there is a computation path for $v^{0..j}$ in A_1 from the start state to a final state, meaning each letter of v from 0 to j makes a valid transition. In other words, for v there exists $\hat{\delta}(p, v, q) \mid p = q0_1, q \in F_1$. Due to the construction, this means that at no step between letter 0 and letter j does v transition to the *fail* state in A_2. Finally, upon reaching an A_1 final state at letter j, word v "falls off" (no longer has a valid transition in) automaton A_2 and henceforth (after letter j) will not activate the *fail* state. Therefore, the checker holds ($v \notin L(A_2)$, *fail* state not visited on input v).
    - Suppose the checker holds on input v ($v \notin L(A_2)$, *fail* state not visited on input v). Similar to the argument above, this means that the execution of v must have "fallen off" A_2 prior to the end of execution, meaning that at the same execution point v enters a final state in A_1. Therefore, $\exists j < |v|$ such that $v^{0..j} \models\!\!\models \text{r}$, so $v \models \text{r!}$.
  - Conclusion: $v \models \text{r!} \Leftrightarrow$ the checker holds ($v \notin L(A_2)$, *fail* state not visited on input v).

- *Weak SERE (Property)*

  - Semantic definition: $v \models \text{r} \Leftrightarrow \forall j < |v|, v^{0..j\top\omega} \models \text{r!}$. Since this definition depends on the previous one, the semantics for r!, we combine the two, yielding the "direct semantics": $v \models \text{r}$ iff ($\forall j < |v|$, $\exists k < \omega$, $(v^{0..j\top\omega})^{0..k} \models\!\!\models \text{r}$) [94]. In other words, $v \models \text{r}$ iff any finite prefix $v^{0..j}$ of v can be $\top$-extended an arbitrary length in order that a k-length finite prefix of it (the extended word) SERE-satisfies r. This allows "in flight" executions that begin to SERE-satisfy r at v^0, but fail to complete by the end of execution, to not be considered failures simply because execution terminated (i.e., they are interpreted in the *weak view* [62]).
  - Automaton construction: As before, we convert a conditional-mode automaton A_1 for SERE r into a fail-mode automaton A_2, to interpret r using the property semantics. In this case, though,

we do not add end-of-execution transitions to the fail state in A_2, as we did with strong SEREs. See Algorithm VII.1 for a description.

- Analysis: Suppose that conditional-mode automaton A_1 models r by SERE semantics. In other words, $v \models\!\!\!\models \mathrm{r} \Leftrightarrow$ (there exists $\hat{\delta}(p, v, q) \mid p = q0_1, q \in F_1$).
  - Suppose $v \vDash \mathrm{r}$. By the (direct) semantics, $\forall\, j < |v|,\ \exists k < \omega,\ (v^{0..j\top\omega})^{0..k} \models\!\!\!\models \mathrm{r}$. Informally, we can say that "for any prefix $v^{0..j}$ of v, plus an arbitrary-length $\top$-extension, there exists a finite prefix (of length k+1) that SERE-satisfies r." Note that k may be greater than j, and in fact k may be greater than $|v|$, as well. Consider a prefix $v^{0..j}$ of v, arbitrarily $\top$−extended. Suppose $k \leq j$. In this case, during computation of v from v^0 to v^k, A_1 remains on a valid path toward acceptance, so A_2 never reaches the fail state. Now suppose $k > j$. In this case, A_1 is on a valid computation path in A_1 until v^j, then remains on a valid computation path in A_1 over $v^{j..k}$, since v is $\top$-extended beyond j. By the construction, A_2 again never reaches the fail state. The checker holds ($v \notin L(A_2)$, *fail* state not visited on input v).
  - Suppose the checker holds on input v ($v \notin L(A_2)$, *fail* state not visited on input v). It must be the case that every j-length prefix of v starting at v^0 itself has a prefix (arbitrarily $\top$-extended, as necessary) with a computation path to a final state in A_1, hence a prefix that supports (by SERE semantics) r; otherwise, the fail state in A_2 would have been triggered by the computation of $v^{0..j}$. Therefore, $\forall\, j < |v|,\ \exists k < \omega,\ (v^{0..j\top\omega})^{0..k} \models\!\!\!\models \mathrm{r}$, so $v \vDash \mathrm{r}$.
- Conclusion: $v \vDash \mathrm{r} \Leftrightarrow$ the checker holds ($v \notin L(A_2)$, *fail* state not visited on input v).

- *Property Abort Boolean.*
  - Semantic definition: $v \vDash p$ abort $b \Leftrightarrow (v \vDash p)$ or ($\exists\, j < |v|$ such that $v^j \vDash b$ and $v^{0..j-1\top\omega} \vDash p$). Informally, v satisfies p abort b iff either v satisfies p, or, if the abort signal b was true at some time during v, and prior to the abort, the prefix of v (arbitrarily $\top$-extended as needed) satisfied p.
  - Automaton construction: Given a fail-mode automaton A_1 for p, to construct A_2 for p abort b, we attach "$\wedge\neg b$" to the condition on each edge, so that any time b is true, the automaton is "reset."
  - Analysis: Suppose A_1 models the property semantics of p. In other words, $v \vDash p \Leftrightarrow (v \notin L(A_1)$, *fail* state not visited on input v).
    - Suppose $v \vDash p$ abort b. By the semantics, either $v \vDash p$ or $\exists\, j < |v|$ such that $v^j \vDash b$ and $v^{0..j-1\top\omega} \vDash p$. In the first case, on input sequence v, the fail-mode automaton never enters the fail state in A_1, which means that it will not do so in A_2, since the added effect of b in A_2 can only cause a reset, but never cause a fail-state entry. In the second case, there exists a cycle in which b is asserted prior to the end of v, at cycle j; but prior to j, $v^{0..j-1}$ never caused A_1 to enter the *fail* state, and hence did not cause A_2 to enter the *fail* state. So, the checker holds ($v \notin L(A_2)$, *fail* state not visited on input v).
    - Suppose the checker holds on input v ($v \notin L(A_2)$, *fail* state not triggered). Similarly, either $v \vDash p$ (and input v "fell off" the fail-mode automata A_1, and hence A_2) or the abort signal b was triggered at some letter j, and v satisfied p at least until letter $j-1$ without causing a failure. Hence, either $v \vDash p$ or $\exists\, j < |v|$ such that $v^j \vDash b$ and $v^{0..j-1\top\omega} \vDash p$, so $v \vDash p$ abort b.

- Conclusion: $v \vDash p$ abort $b \Leftrightarrow$ the checker holds ($v \notin L(A_2)$, *fail* state not visited on input v).

- *Property Conjunction*.

  - Semantic definition: $v \vDash p_1 \&\& p_2 \Leftrightarrow (v \vDash p_1$ and $v \vDash p_2)$.
  - Automaton construction: Given conditional-mode automata A_1 and A_2, respectively, we convert them each to fail mode. Then, using the automata disjunction method, we combine them into a single automaton, A_3. As mentioned previously, the reasons this works semantically is similar to the function of DeMorgan's laws; the property $p_1 \&\& p_2$ holds only if both p_1 and p_2 hold, and so $p_1 \&\& p_2$ fails if either p_1 fails or p_2 fails.
  - Analysis: Suppose A_1 models the property semantics of p_1 and A_2 models the property semantics of p_2. In other words, $v \vDash p_1 \Leftrightarrow (v \notin L(A_1)$, *fail* state not visited on input $v)$, and $v \vDash p_2 \Leftrightarrow (v \notin L(A_2)$, *fail* state not visited on input $v)$.
    - Suppose $v \vDash p_1 \&\& p_2$. By the semantics, $v \vDash p_1$ and $v \vDash p_2$. Therefore, on input v, neither A_1 nor A_2 enters its *fail* state. By the (disjunction) construction, A_3 cannot enter a *fail* state either. Hence, the checker holds ($v \notin L(A_3)$, *fail* state not visited on input v).
    - Suppose the checker holds on input v ($v \notin L(A_3)$, *fail* state not visited on input v). Similarly, if A_3 does not enter a *fail* state on input v, then neither does A_1 nor A_2. So $v \vDash p_1$ and $v \vDash p_2$, hence $v \vDash p_1 \&\& p_2$.
  - Conclusion: $v \vDash p_1 \&\& p_2 \Leftrightarrow$ the checker holds ($v \notin L(A_3)$, *fail* state not visited on input v).

- *SERE Suffix-Implication Property*.

  - Semantic definition: $v \vDash \mathrm{r} \mapsto p \Leftrightarrow (\forall\, j < |v|$ such that $\overline{v}^{0..j} \models\!\!\!\equiv \mathrm{r}$, $v^{j..} \vDash p)$ (note the mixed use of SERE satisfaction and property satisfaction in the PSL definition).
  - Automaton construction: Given a conditional-mode automaton A_1 for r, and an fail-mode automaton A_2 for property p, construct automaton A_3 using the fusion algorithm, in effect generating $A_3 = A_1 : A_2$.
  - Analysis: Suppose A_1 models the SERE semantics of r and A_2 models the property semantics of p. In other words, $v \models\!\!\!\equiv \mathrm{r} \Leftrightarrow$ (there exists $\hat{\delta}(p, v, q) \mid p = q0_1, q \in F_1)$, and $v \vDash p \Leftrightarrow (v \notin L(A_2)$, *fail* state not visited on input $v)$.
    - Suppose $v \vDash \mathrm{r} \mapsto p$. By the semantics, $\forall\, j < |v|$ such that $\overline{v}^{0..j} \models\!\!\!\equiv \mathrm{r}$, $\quad v^{j..} \vDash p$. First, we note this takes the form of an implication, and the fused automaton A_3 is in fail-mode. If there exists no prefix of v which meets the antecedent condition (SERE support for r, modeled by A_1), the semantics hold vacuously. In this case, A_3 would never transition from an A_1 state to an A_2 state via the fused edges, so the *fail* state of A_3 cannot be entered. In the cases where a j-length prefix of v does satisfy the SERE semantics for r, it triggers a transition via a fused edge from the A_1 states to the A_2 states. From here, though, the remaining suffix of v, beginning with the jth letter, satisfies the property semantics of p, and does not have a computation path to the *fail* state of A_2 (hence, no computation path to the *fail* state of A_3). The checker holds ($v \notin L(A_3)$, *fail* state not visited on input v).

- Suppose the checker holds ($v \notin L(A_3)$, *fail* state not visited on input v). The fail state of A_3 has not been triggered. There must not exist a computation path on v from the start state of A_1 to the *fail* state of A_2. Therefore, either there is no j-length prefix of v for which the SERE semantics of r hold (and $v \vDash \mathrm{r} \mapsto p$ vacuously), or, for all j-length prefixes of v that satisfy the SERE semantics of r, the suffix of v beginning at letter j satisfies the property semantics of p. Hence $\forall\, j < |v|$ such that $\overline{v}^{0..j} \models \mathrm{r}, \quad v^{j..} \vDash p$, so $v \vDash \mathrm{r} \mapsto p$.

- Conclusion: $v \vDash \mathrm{r} \mapsto p \Leftrightarrow$ the checker holds ($v \notin L(A_3)$, *fail* state not visited on input v).

Summary: We have given a structured argument, for each of the base-case automata implementation strategies, that the construction we use models PSL's formal semantics.

f. Automata to HDL Conversion

For the third stage of the argument, we review the automaton-to-HDL conversion process, which was described in Chapter VII, and note the following:

- Each state in the automaton is modeled by a signal, whose value is updated on each transition of the automaton.
- Any time a state in the automaton is occupied, through valid transitions from the start state, its equivalent signal in the HDL representation is *high*, or "1."
- In a conditional mode automaton, any time a final state of the automaton is occupied, the HDL representation's *hold* output signal is *high* and the *fail* output signal is *low*; when no final state is occupied, *hold* is *low* and *fail* is *high*.
- In a fail mode automaton, any time the "fail" state is occupied, the HDL representation's *fail* output signal is *high* (and *low* otherwise), and its *hold* signal is *low* (and *high* otherwise).
- The automaton and its HDL equivalent are started (start state activated) on the same conditions, and each make one move per clock cycle.

By its construction, the HDL representation matches the semantics of the checker automata. For conditional mode automata, the checker signals a hold on any clock cycle when a final state is active, and signals a fail otherwise. For fail mode automata, the checker signals a fail on any clock cycle in which the fail state is active, and signals a hold otherwise. We conclude by inspection that the HDL implementation of a checker mirrors the interpretation of the automata representation of the checker.

g. Summary

Through the combination of arguments in stages one through three, above, we conclude: "For any PSL formula in the Simple Subset, the method creates an HDL assertion-checker whose semantics over input sequences are equivalent to those of the original PSL formula. For a given input sequence, the assertion-checker *accepts* an input sequence if and only if the input sequence *satisfies* the semantics of the PSL formula on which the checker was based."

B. ASSERTION CHECKER AUTOMATA AND MODEL CHECKING AUTOMATA

In this section, we contrast the use of automata in assertion checkers, as employed in this research, with the use of automata and other finite state machines (FSMs) in model checking, which was briefly described in Chapter V.

1. Finite State Machines for Representing Kripke Structures

According to Clarke, the model checking problem can be stated as follows:

> Let M be a Kripke structure (i.e., state-transition graph). Let f be a formula of temporal logic (i.e., the specification). Find all states s of M such that $M,s \vDash f$. We use the term *model checking* because we want to determine if the temporal formula f is *true* in the Kripke structure M, i.e., whether the structure M is a *model* for the formula f. [43]

Given a set of propositional variables P, a Kripke structure [104] K is defined as $K = \{S, I, R, L\}$, where:

- S is a finite set of *states*
- $I \subseteq S$ is a set of *initial states*
- $R \subseteq S \times S$ is the *transition relation* between states
- $L : S \longrightarrow 2^P$ is a *valuation function*, which maps each state to a set of variables

and:

- A *path* of the structure is a sequence of states $\pi = s_0 s_1 s_2 s_3 \ldots$ such that $s_0 \in I$, and $\forall i \geq 0$, $(s_i, s_{i+1}) \in R$.
- A *word* on the path is a sequence of sets of variables such that $v = L(s_0), L(s_1), L(s_2) \ldots$

Note the resemblance to the definition of a PSL Model, given in the previous section, from the PSL specification. Though not explicitly stated in the PSL specification, it is clear that a PSL Model, as defined there, is a Kripke structure. In the context of hardware verification, the possible values of the hardware circuits in the design define the states of the system being modeled by the structure, and the valuation function maps each state to a set of variables (i.e, the current assignments of the variables in P).

A good description of Kripke structures and their relationship to FSMs is given by K. Schneider:

> Kripke structures are closely related to finite state automata, and in fact, Kripke structures may be viewed as Moore machines that read letters from a singleton alphabet (the label function is the output function). Note that a Kripke structure only defines the states and computations of a system, but it does not provide any form of *causality*. This means that Kripke structures do not explain why the system is in a specific state, or why it moves to another state. In particular, Kripke structures do not distinguish between inputs, outputs, program locations, and local variables. Instead, they collect the possible values of the different variables that can occur in the computations of a system. [104]

The following is an example of a finite state machine that is defined by a Kripke structure, which might be used in model checking:

- $S = \{s_0, s_1, s_2, s_3\}$
- $I = \{s_0\}$
- $R = \{(s_0, s_2), (s_2, s_3), (s_3, s_1), (s_1, s_3), (s_1, s_0)\}$
- $L = \{(s_0, \{x,y,z\}), (s_2, \{x,y\}), (s_3, \{y,z\}), (s_1, \{z\})\}$

As mentioned earlier, the shorthand notation of listing a propositional variable's name indicates that it has a value of *true* in that state. For example, we could have written the first valuation in L as (s_0, {*x=true, y=true, z=true*}). The FSM representation is depicted in Figure 43.

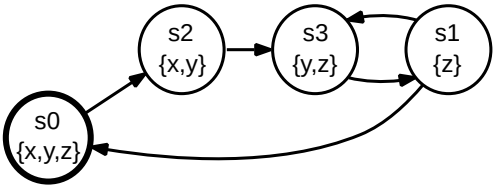


Figure 43: Example finite state machine representation of a Kripke structure.

A possible computation *path* in this structure is $\pi = s_0 s_2 s_3 s_1 s_3$. The output *word* associated with the path is $v = (\{x,y,z\}, \{x,y\}, \{y,z\}, \{z\}, \{y,z\})$.

In Chapter VII, we describe the automata used in the assertion-checker method outlined by Boulé and Zilic. These automata differ from the FSMs representing Kripke structures, as used in model checking. The principal difference is that the Kripke structure FSMs model a *system* (e.g., a hardware design), whereas the assertion-checker automata model the computation of a particular input sequence on a PSL *formula*. When a checker automata computes a particular input sequence, its computation paths represent various ways the input may lead to acceptance (in the case of conditional-mode automata) or rejection (in the case of fail-mode automata) by the checker for that formula.

Another way of viewing the distinction is that the Kripke FSMs produce sequences of propositional assignments as an *output,* whereas the checker automata consume a sequence of propositional assignments as *input*. In addition to these differences, there is the superficial difference that the Kripke structure FSMs have no transitions on their edges and no accepting states, both of which the checker automata have.

2. Automata for Model Checking vs. Automata for Dynamic Assertion Checking

Researchers have used several different types of automata in model checking. For example, Büchi Automata have been used in model checking PSL [105], [106]. Similarly, Alternating Büchi Automata were used for verification on a subset of PSL by Ben-David et al. [107]. Boulé and Zilic note that, besides Büchi

Automata and Alternating Büchi Automata, other types, including Alternating Automata, Universal Automata, and Existential Automata, have all been used in various static and dynamic verification research [6].

Consider Büchi automata, as a common example. A nondeterministic Büchi automata [107] may be defined as a six-tuple $B = (\Sigma, S, I, \rho, F, A)$, where:

- Σ is a nonempty finite alphabet
- S is a nonempty finite set of states
- $I \subseteq S$ is a nonempty set of initial states
- $\rho : S \times \Sigma \rightarrow 2^S$ is a transition function
- $F \subseteq S$ is a set of final states
- $A \subseteq S$ is a set of accepting states

Büchi Automata have the advantage of defining acceptance for both finite and infinite inputs. They do so by distinguishing *accepting* states and *final* states, which may be considered synonymous terms in classical automata [75]. In a Büchi Automata, an *infinite* run is accepting if an *accepting state is visited infinitely often*; a *finite* run is accepting if the last state is a *final state*, just as in classical automata [107]. Büchi Automata have been employed to represent LTL[20] formulas, and in fact efficient algorithms exist for automatically translating an LTL formula into its equivalent Büchi Automata representation [37].

Despite the utility of Büchi Automata and other automata types in model checking, Boulé and Zilic conclude that these more general automata forms are not necessary in their PSL checker generator method, and that propositional logic automata, or PLAs, are sufficient. Boulé and Zilic offer this comment as one reason for not adopting an automata type from the model checking field:

> The run-time semantics exhibited by the automata developed for model checking do not offer the run-time behavior we wish to support in our checkers for dynamic verification, where *assertion errors are to be reported in real time throughout the execution trace* [emphasis added]. [6]

In addition to the error reporting issue, we observe that the definitions in Büchi Automata that permit evaluation of infinite sequences are not necessary in dynamic verification, where we are concerned with evaluating only a single, finite execution trace at a time.

In short, though other types of automata do appear useful for model checking, the simpler propositional logic automata method of Boulé and Zilic is sufficient for evaluating the semantics of PSL formulas, using synthesizable checkers, against the dynamic execution of a hardware design.

C. OVERHEAD ESTIMATION

In order to approximate the total overhead incurred by our method in a general design, we first obtained statistics on average checker size. We gathered 65 total assertions, including those published as PSL

[20] As noted in Chapter VI, PSL is derived, in part, from LTL.

assertion benchmarks by Boulé and Zilic [71], and Abarbanel et al. [5], plus the PSL assertions we used in the OpenRISC experiments, and ran them all together through our checker-generator, psl2hdl. The results are summarized in Table 17, which shows the mean and maximum number of states and edges in the generated automata.[21]

	Mean	**Maximum**
States	7.2	25
Edges	10.9	39

Table 17: Generated automaton size metrics for a set of 65 benchmark assertions.

Each checker automaton does not incur much physical space in its hardware representation. Each state of the automaton is modeled as a single-bit flip-flop. Each outgoing edge incurs a single AND-gate, and there is an OR-gate fan-in at each state's input. Consider the example in Figure 44. In the automaton, at left, state $q2$ is reachable from either state $q0$ or state $q1$. The automaton representation in hardware permits the flip-flop representing state $q2$ to be active (high) in clock cycle *n+1* if either $q0$ was active at clock cycle n and expression b was *true*, or if $q1$ was active at clock cycle n and expression c was *true*. In a Verilog-style "nonblocking"[22] assignment, this looks like: "$q2 <= (q0\&\&(b)) \,||\, (q1\&\&(c))$;" which assigns the value of $q2$ for the next cycle, based on the values in the right-hand side of the expression during this clock cycle. Note that some OR-gates, like the one on the input to $q2$, may need to be fan-in OR-gates if there are many edges, occupying more space than a normal two-input OR-gate.

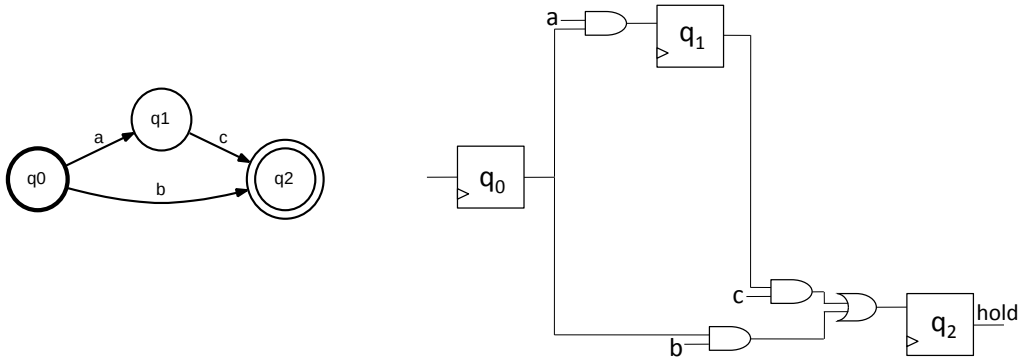


Figure 44: Checker automaton example (a), and its circuit equivalent (b).

The logic resources required to implement an automaton in circuit form are listed in Table 18.

[21] Trivial automata, such as those accepting the empty set or the empty input sequence, are not factored into the calculations.
[22] Informally, a concurrent assignment evaluated whenever its surrounding code block is triggered.

Resource	Maximum Number Required
Flip-Flop	$\|Q\|$
AND-gate	$\|\delta\|$
OR-gate	$\|Q\|$

Table 18: Logic resources required to implement an automaton in circuit form.

Based on the sample data, then, we conclude that an *average* assertion-checker, based on a Simple Subset PSL assertion, requires approximately eight flip-flops and 19 logic gates to implement in a circuit, plus input and output buffers for each of the named circuits, a clock circuit connecting to each gate, and a reset signal that connects to all states.

Recall from Chapter VII that it may be necessary to duplicate assertion checkers, if (functionally equivalent versions of) the signals they monitor cascade hierarchically through a design. It is difficult to quantify how this repetition will affect the total number of checkers required; however, we can take a cue from the hierarchical depth of a general-purpose processor design. In the case of MINSOC, the design units are not nested deeper than six levels. Because the MINSOC design is not as complex as modern commercial processors, we assume that a processor module hierarchy could be as deep as approximately ten levels, or perhaps slightly more. Hence, we approximate the impact of needing to duplicate functionally equivalent checkers through the hierarchy as a single order of magnitude, though it will often be much less.

To obtain an approximation of the number of behavioral restrictions one might find in a more mature, more completely documented processor model, we examined the three volumes of the MIPS architectural manual [108]. In them, we identified 52 general behavioral restrictions, which relate to the function of the privilege modes, registers, exceptions, access control units like TLBs and MMUs, etc. If we add in behavioral restrictions that are particular to how a specific instruction should be handled, the total is closer to 100. The exact number depends on how you count them, since a number of the restrictions are broken down into sets of sub-restrictions.

If we take a conservative approach, and assume that the number of behavioral restrictions could be as much as an order of magnitude greater, given the increasing complexity of modern designs, and the fact that many processor architectures have optional components which may be added, we can imagine that as many as 1,000 individual restrictions might need to be mapped in a well-covered general-purpose processor. Given our earlier assumption that hierarchical redundancy might add another factor of 10, the total number of assertion checkers may be on the order of 10,000, in a general-purpose processor. If we multiply this by the average number of flip-flops and gates for a typical checker, the conservatively estimated overhead is on the order of 100,000 flip-flops and 200,000 logic gates. It takes several transistors to form logic gates and flip-flops, depending on the implementation technology, but as a point of reference, modern commercial general-purpose processors contain several billion transistors [109], so the checker overhead does not dominate the overall size of the design, according to these rough estimates.

D. ALGORITHMIC COMPLEXITY

In this section, we examine the time and space required to perform some of the algorithms employed in our method. The time cost reflects the time to generate the checkers, and the space reflects the output automaton size (and hence, HDL module size) of the checkers.

1. Rewrite Rules

If a PSL input formula has length n, as defined by the number of lexical elements (leaves on its parse tree), we know that the number of PSL operators (such as $\mapsto$, `next_event`, `always`, etc.) is less than n. The rewrite rules can trigger the application of further rewrite rules, but due to the analysis in Section A, we know that the chain of rewrites never exceeds eight instances (the maximum height of the graph in Figure 41), and that each individual rewrite takes constant time, with two exceptions.

The exceptions are the third and fourth rewrite rules for SEREs (`r[*c]` $\Longrightarrow$ `r;r;r...r` (*c times*), and `r[*l:h]` $\Longrightarrow$ `r[*l]|...|r[*h]`), which handle repetition. Each of these can cause an increase in the size of the rewritten formula by a factor of c, (or h-l, respectively). We denote, for a formula, the overall repetition factor as m, representing the summation of all the individual repetition factors, like c and h-l. In the unusual case where repetition operators are *nested* within each other, we must multiply any nested repetition factors; for example, in the SERE "`{a;b[*3]}[*5]`" the nesting of quantifiers leads to an m value of 15.

Combining these factors, the overall time and space required of the rewrite algorithm is $O(mn)$.[23]

2. Automata Construction

Next, we examine how much time and space are required by the algorithms for converting a rewritten PSL parse tree into an equivalent automaton. Our overall algorithm for composing the automaton employs a depth-first walk through the PSL tree.[24] For a graph (in this case, the PSL parse tree) with V vertices and E edges, the time required for depth-first traversal is *O(V+E)* [110]. The overall time cost will be this amount multiplied by the time required to perform the automata computations at each node.

The simplest cases require construction of automata to accept the empty set, the empty input sequence, or a single instance of a boolean expression. Each of these is constructed in constant time and space (see Figure 18). These cases will occur at the leaves of the input PSL tree; the others, which follow, employ some type of combination, and take place at non-leaf nodes.

Suppose we have two input automata, A_1 and A_2, and we wish to combine them using the concatenation algorithm. We denote the number of states using $|Q_1|$ and $|Q_2|$, and the number of edges using $|\delta_1|$ and $|\delta_2|$. Using the application of the concatenation algorithm described in Chapter VII, the time required to execute the algorithm and the size of the constructed automaton are both $O(\,|Q_1|+|Q_2|+|\delta_1|+|\delta_2|\,)$.

[23] Since their primary publications on SEREs and Properties, Boulé and Zilic have recently described expressing SERE repetition more efficiently by modeling it directly in hardware [6]. A similar approach is also mentioned by Findenig [4]. Because traditional automata do not provide a way to explicitly represent stored data (state), we adhere to the original approach, where repetition is handled using the rewrite rules (resulting in a larger rewritten syntax tree). A comparison of the various approaches for handling SERE repetition would be useful future work.

[24] It is not clear whether Boulé and Zilic also employ a classical depth-first traversal algorithm, though they describe their composition as "recursive" in nature [6].

Automata fusion is similar to concatenation, but we have to produce the Cartesian product of (the incoming edges to final states in A_1) and (the outgoing edges of the start state in A_2). The number of states in the resulting automaton is $O(\ |Q_1| + |Q_2|\ )$, while the number of edges is $O(\ |\delta_1| \times |\delta_2|\ )$, with a resulting total time required of $O(\ |Q_1| + |Q_2| + |\delta_1| \times |\delta_2|\ )$. The number of transitions, $|\delta|$, in an automaton, will always be at least $|Q| - 1$ in our case (because unreachable states are always removed); therefore, the product term dominates and we can simplify the overall time to $O(\ |\delta_1| \times |\delta_2|\ )$.

Automata disjunction, also computationally simple, is $O(\ |Q_1| + |Q_2| + |\delta_1| + |\delta_2|\ )$ in time and space required.

Automata conjunction, used for length-matching intersection, is similar in PSL to the product construction for classical automata [75]. However, as previously noted, the use of PLAs introduces some additional requirements. Though our algorithm considers only reachable states and is therefore on average more efficient than a brute-force approach, we consider the latter as a worst case upper bound, which would be approximately the case if both A_1 and A_2 are fully connected graphs (every node has an edge to every other node). The upper bound on the number of resulting states is $O(\ |Q_1| \times |Q_2|\ )$. For each of the resulting states (p,q), the maximum number of outbound edges that might need to be considered is the cross-product of the maximum number of possible outgoing edges from p and q, which in a fully connected graph is again $|Q_1| \times |Q_2|$. Since this is the number of edges to be considered for each resulting state, the total computational requirement for the resulting states and edges is a time of $O(\ |Q_1| \times |Q_2| + |Q_1|^2 \times |Q_2|^2\ )$, or just $O(\ |Q_1|^2 \times |Q_2|^2\ )$. Since the resulting automaton could also be fully connected, the resulting upper-bound space requirement, counting both states and edges, is the same.

The Kleene closure procedure loops the edges inbound to an automaton's final states back to the start state. It requires O($|Q| + |\delta|$) time. The states in the resulting automaton are the same, or $O(\ |Q|\ )$, and the edges will no more than double in number, and hence remain $O(\ |\delta|\ )$.

Given an automaton A for accepting sequences described by a property p, we can abort any in-flight sequences with the "`p abort b`" implementation. Since it requires only modifying all the existing edges with the addition of "$\wedge\neg b$," the resulting automaton has no new states or edges, and the time required to perform the construction is $O(\ |\delta|\ )$.

Three property base-case implementations require the construction of at least one fail-mode automaton: property conjunction (`p1 && p2`), strong form of a sequence (`s!`), and sequence suffix-implication property (`s` $\mapsto$ `p`). Since the automata may otherwise be deterministic or nondeterministic, but must be deterministic in fail mode, the fail-mode conversion requires determinization. Given an input automaton with $|Q|$ states, the maximum number of states in the resulting automaton is $O(\ 2^{|Q|}\ )$. Though not normally observed in practice, this potentially exponential increase can be a significant limiting factor in the application of this method. We are not aware of an automata-based method for accepting PSL formulas that avoids the semantic requirement for determinization, as described by Boulé and Zilic [6]. For these three cases, the determinization factor eclipses the other computational requirements, leading to an algorithmic upper-bound time and space of $O(\ 2^{|Q|}\ )$. In the case of property conjunction, which has two input automata, we use the larger Q of the two. Though not likely, such an automaton, if fully connected, would have $O(\ 2^{|2Q|}\ )$ edges.

The algorithmic complexity of the constructions are listed in Table 19.

	Method	Description	Output States	Output Edges	Conversion Time
1	Empty set, empty sequence, boolean	Ø,`[*0]`,`b`	$\theta(c)$	$\theta(c)$	$\theta(c)$
2	Concatenation	`r1;r2`	$O(\|Q_1\|+\|Q_2\|)$	$O(\|\delta_1\|+\|\delta_2\|)$	$O(\|Q_1\|+\|Q_2\|+\|\delta_1\|+\|\delta_2\|)$
3	Disjunction	`r1||r2`	$O(\|Q_1\|+\|Q_2\|)$	$O(\|\delta_1\|+\|\delta_2\|)$	$O(\|Q_1\|+\|Q_2\|+\|\delta_1\|+\|\delta_2\|)$
4	Fusion	`r1:r2`	$O(\|Q_1\|+\|Q_2\|)$	$O(\|\delta_1\|\times\|\delta_2\|)$	$O(\|\delta_1\|\times\|\delta_2\|)$
5	Length-Matching Intersection	`r1&&r2`	$O(\|Q_1\|\times\|Q_2\|)$	$O(\|Q_1\|^2\times\|Q_2\|^2)$	$O(\|Q_1\|^2\times\|Q_2\|^2)$
6	Kleene Closure	`r[*]`	$O(\|Q\|)$	$O(\|\delta\|)$	$O(\|Q\|+\|\delta\|)$
7	Property Abort Boolean	`p abort b`	$O(\|Q\|)$	$O(\|\delta\|)$	$O(\|\delta\|)$
8	Property: Sequence Strong Form	`s!`	$O(2^{\|Q\|})$	$O(2^{\|2Q\|})$	$O(2^{\|2Q\|})$
9	Property Intersection	`p1&&p2`	$O(2^{\|Q\|})$	$O(2^{\|2Q\|})$	$O(2^{\|2Q\|})$
10	Suffix-Implication	`s↦p`	$O(2^{\|Q\|})$	$O(2^{\|2Q\|})$	$O(2^{\|2Q\|})$

Table 19: Algorithmic complexity of automata constructions.

Recall that depth-first traversal of a rewritten parse tree with V nodes and E edges will take $O(V+E)$ time. Since the construction methods are sometimes polynomial in the size of the input automata and sometimes exponential, we break the construction into two cases: those formulas *requiring* a fail-mode automaton and those *not requiring* one.

In the first case, only the first seven methods listed in Table 19 will be employed. At worst, the highest time-cost rule (length-matching intersection) will be used at every node. Even though we can potentially square the number of automata states at each level of nesting the formula, the result of each step will always be a polynomial in Q. The polynomial degree will depend on how deeply the formula's operators are nested.

In the second case, a fail-mode conversion is called for at some point, and so at least one step in the formula's automaton construction incurs the use of an asymptotically exponential algorithm, such as one of the last three in Table 19. Though the number of states observed in practice is not normally that large, it is conceivable that the exponential construction can lead to automata that are too large to practically implement, or too time-consuming to compute, in generating an assertion checker.

In Algorithm VII.3, in Chapter VII, we outline a DFA minimization procedure for PLAs. The boolean simplification step (if needed), in line 12, requires a time factor that is linear in the length of the newly formed boolean expressions. To obtain an upper bound, let δ_{max} represent the total number of terms in the conjunction of the two longest boolean expressions on edges in all of δ. A time factor of Q is introduced in each of lines 7, 8, 9, and 11, resulting in an overall upper-bound time complexity of $O(\delta_{max}\cdot|Q|^4)$. The upper bound on space used by the algorithm is defined by the distinguishability matrix, of size $O(|Q|^2)$.

3. Automata to HDL Conversion

See Algorithm IX.1 for a representation of the conversion steps, given as input a propositional-logic automaton with $|Q|$ states, $|\delta|$ edges, and $|\Pi|$ distinct propositional variables.

Algorithm IX.1 Automaton to HDL conversion steps.

	Algorithm Step	Time
1.	For each variable $\pi \in \Pi$, add an input signal to the module.	$\theta(\lvert\Pi\rvert)$
2.	Add inputs for the *clock* and *reset* signals, and add outputs for *hold* and *fail*.	$\theta(c)$
3.	Add a local signal for each state $q \in Q$ in the automaton.	$\theta(\lvert Q\rvert)$
4.	For each state $q \in Q$:	$\theta(\lvert Q\rvert + \lvert\delta\rvert)$
5.	Construct an assignment statement that represents the disjunction of all incoming edges to q. Each incoming edge is represented by the conjunction of the signal for its originating state, and the boolean expression defining its edge condition.	
6.	Create assignments for the *fail* and *hold* signals:	
7.	If the automaton is in conditional mode, the *hold* signal is represented by the disjunction of the signals for all final states (and the *fail* signal is its negation).	
8.	If the automaton is in fail mode, the *fail* signal is represented by the final state labeled "fail" (and the *hold* signal is its negation).	

We conclude that, given an input automaton, the time and space required by the algorithm for converting it to an HDL representation is $\theta(|Q| + |\delta| + |\Pi|)$.

4. Summary

Most algorithms in our checker-generation process use time and space that are polynomial in the input formula size. However, several commonly-used cases, e.g., those incurring the construction of fail-mode automata, employ algorithms with exponential time and space complexity. In practice, the input formulas are normally small enough that this does not preclude our ability to generate synthesizable checkers.

E. STRENGTHS AND LIMITATIONS

We assess the following as the strengths and limitations of the method.

Strengths:

- Early Frame of Reference. Security checkers do not rely on a trusted "golden sample," whether it is a trusted high-level design or a trusted processor sample. Though equivalence-checking techniques are valuable, their reference artifact is itself a form of intermediate implementation, rather than an original specification. By using an early frame of reference like the architectural specification and other original design documents, our "baseline," against which a processor's behavior is compared, is not subject to intermediate subversion, as is the case with equivalence-checking methods. This is illustrated in Figure 45.

- Persistence. One advantage of evaluating security dynamically, rather than statically, is that static analysis of a high-level design does not account for malicious changes made afterward, to the low-level design or to the physical system. Dynamic security evaluation, as by an EM, on the other hand, can be used to detect MIs inserted after the high-level design phase, in prototypes or even in fielded systems. Other design analysis methods may only detect MIs inserted prior to, or during, the high-level design phase, not afterward. However, a sophisticated attacker, subverting a design that has security checkers built in, could emplace an MI *and* bypass the monitoring system at the same time, of course, but in this case the presence of the monitoring mechanism at least adds a degree of difficulty to the task.

- Portability. Because they are constructed of synthesizable HDL units, security checkers, once created, can be added to a processor using any hardware-design platform, and they can be exercised in any HDL simulator, even one which has no native assertion support.

- Size. Physical analysis methods are limited in their ability to detect changes in a processor whose size is a small fraction (around .01%) of the overall processor size, or smaller [1], [111]. Design analysis methods, like the use of security checkers, can detect much smaller changes to an RTL design.

- Compatibility. Runtime security checkers can be used in concert with a variety of other techniques. For example, after designing security checkers for a processor, the processor design netlists can be obfuscated, to deter subsequent tampering, without affecting operation of the monitor units. Also, as described earlier, design analysis methods that identify rarely-used or unused circuits can be applied with security checkers in a complementary fashion.

Limitations:

- Level of Effort. The most significant limitation to this method is the level of effort it requires. Creating security checkers for an entire design imposes a great deal of work on the designer, and adds to the normally significant effort of ordinary functional verification, which is closely related. The methodology also requires architectural designers to more fully elucidate permitted vs. proscribed behaviors, compared to the level of detail normally used today. The creation of security checkers for a design requires a fair amount of implementation-specific knowledge, and because the effort has to be repeated for each new implementation of an architecture, cannot be amortized. To reiterate our observation from Chapter VI, "a clear and complete statement of any behavioral restrictions in the architecture is necessary for successful application of our method."

- The method does not demonstrate efficient modeling of the correct function of stateful processor elements, such as RAM. We can model the correct operation of every unitary storage element with its own checker, but this will be inefficient. Our method can be applied to the control mechanisms of a storage unit (i.e., load and store controllers), but may not detect the subversion of individual memory cells.

- Unspecified, non-malicious additional circuitry. It is possible that an attacker may add circuitry that is not part of an architecture to a processor implementation, and that circuitry does not cause any specified behavioral restrictions to be violated. It would be useful to be able to detect the presence of

such additional circuitry, even if it causes no violations. However, our method does not detect the extra circuits unless they interact with the specified circuits in a way that causes a violation.

- If an adversary gains access to the processor after the checkers have been added, it is possible that both the processor and the checkers can be subverted, though this will be more challenging for the attacker than if the checkers were never added.

- Enforceability of All Properties. As demonstrated in our example, liveness properties may not always be enforceable at runtime, and disablement attacks are easy to implement and difficult to defend against. Security checkers, in general, will be limited to enforcing safety properties. This limitation is expected, due to the constraints of EM, as outlined by Schneider [34].

In Figure 45, adapted from Bilzor et al. [65], we depict graphically the development lifecycle of a processor, and indicate at which point various security techniques are applied. In the diagram, our technique is called Security Checkers, in the rightmost column. The techniques for the other three columns are described in Chapter V. We use the term *processor reference defined* to indicate the stage which provides the reference design against which samples are tested; *method application stage* to identify where in the processor development lifecycle the MI-detection technique is applied; and *attacks potentially detected* to indicate those phases which, should an MI be inserted then, the particular MI-detection technique *may* be able to recognize it. We believe it is ideal for a method's *processor reference defined* phase to be as early as possible, so that the reference itself is least susceptible to subversion, and that the *method application stage* and *attacks potentially detected* windows should each cover as much of the development lifecycle as possible. As illustrated, our method, called Security Checkers in this graphic, uses an earlier processor reference, and has larger method-application and potential-detection windows, compared to other MI-detection methods.

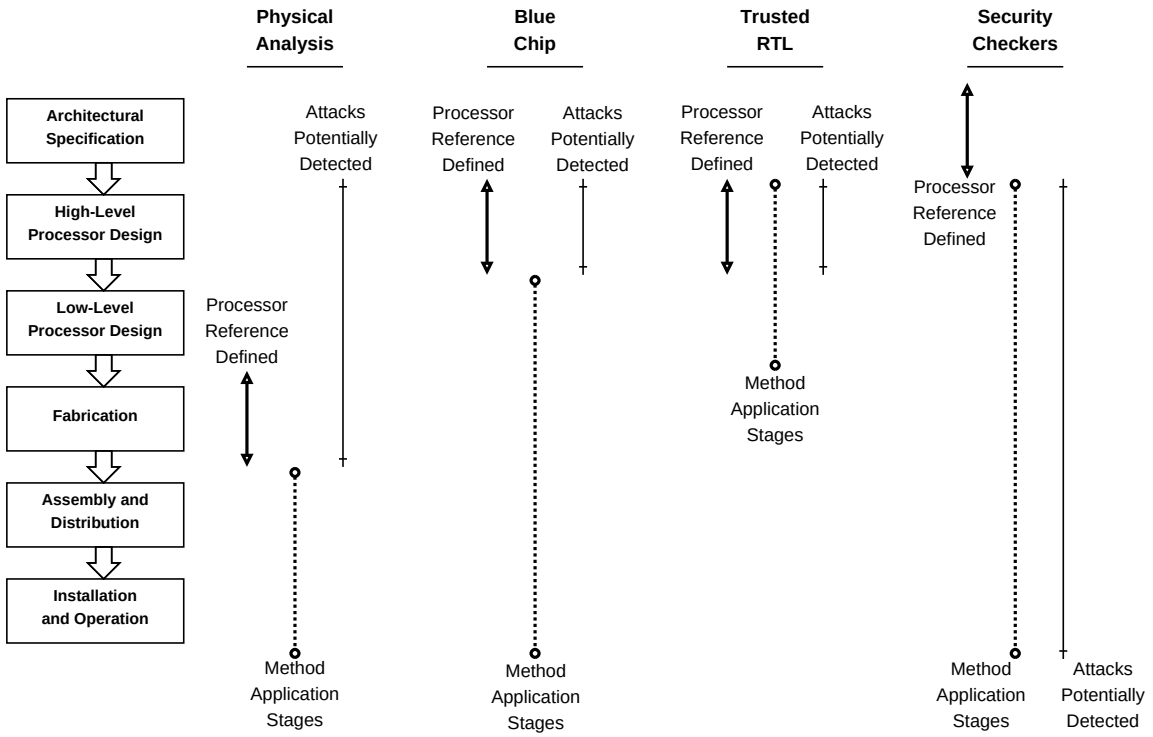


Figure 45: Lifecycle phases for *processor reference defined*, (earlier is better), *method application stages* (larger is better), and *attacks potentially detected* (larger is better), for various MI-detection methods.

F. SUMMARY

In this chapter, we estimate the overhead of our assertion checkers and give arguments for the soundness and completeness of the method overall, as well as the soundness and completeness of the PSL-to-automaton conversion in particular. We show that all PSL Simple Subset formulas describe sets of input values which are regular, and we show the algorithmic complexity for the various cases encountered in the conversion process. Finally, we assess the method's strengths and weaknesses.

THIS PAGE INTENTIONALLY LEFT BLANK

X. CONCLUSIONS AND RECOMMENDATIONS FOR FUTURE WORK

A. SUMMARY

We began our investigation with several questions:

1. How can we characterize the expected security threat to general-purpose processors?
2. What does it mean to say that a hardware design is *secure*?
3. How does hardware security differ from software security?
4. Against what standard can the security of a processor implementation be judged?
5. What techniques currently exist for examining the security of a processor, and what are their strengths and limitations?
6. In what manner might we formulate and express hardware security requirements, so that they can be verified to hold or not hold in a particular implementation?
7. By what mechanism can we perform such evaluation dynamically, in real time?
8. Is there a method by which we can consistently test the same security requirements across the hardware development lifecycle?
9. Is it possible to detect hardware malicious inclusions by observing violations of behavioral requirements in a processor?
10. If we implement runtime checkers for dynamic evaluation of security requirements in hardware, can we do so efficiently, and are there cases where the overhead cost may be excessive? Also, is the method sound and complete? Are there cases it will not cover?

In Chapter I, we frame these questions and our research goals. In Chapters II and III, we answer Question 1 with a detailed analysis of the published academic and real-world examples of Hardware Trojans, or hardware malicious inclusions (MIs). We look at their characteristics, to shape our expectation of future threats.

In Chapter IV, we discuss possible ways to answer Questions 2-4, characterizing the difference between the levels of abstraction expressed in software and hardware, and their respective security requirements. We conclude that the lower abstraction level intrinsic to hardware designs necessitates an expression of security requirements in a similar level of abstraction, and suggest the suitability of *behavioral restrictions* for that purpose, instead of traditional, abstract, software-style security policies.

In Chapter V, we answer Question 5 by performing a survey of existing hardware security methods, grouping them into *physical analysis* and *design analysis* categories.

In Chapter VI, we address Question 6 with a review of the Property Specification Language for expressing hardware behaviors, and an explanation of its semantics.

In Chapter VII, we answer Questions 7 and 8 by describing the method of Boulé and Zilic for creating synthesizable assertion checkers for PSL [6]. We explore the development of assertions from text-based behavioral requirements in an architectural document, then show how the assertions can be converted into dynamic checkers in HDL form, and added to a hardware design. We discuss how the generation of synthesizable assertion checkers allows us to employ them in simulation, FPGA emulation, and fabricated 2D and 3D silicon designs. We review the function of our own software checker-generator, created for this research, and compare it to similar tools.

In Chapter VIII, we answer Question 9 using a set of simulation experiments on an open-source system-on-chip design. We design three malicious inclusions with rare-event triggers, and show in simulation how they might be detected if they do violate certain specified behavioral restrictions, as expressed by the assertion checkers. We show that, even if the malicious inclusions are not triggered, we can use code coverage analysis as a complementary technique to help identify potentially malicious circuits, greatly reducing the portion of a design that needs to be analyzed manually. We also verify that the synthesizable hardware assertion checkers created by our tool agree semantically with the same (software) assertion checkers in our commercial simulator.

In Chapter IX, we answer the set of queries in Question 10. We discuss cases in which malicious behavior in a processor will and will not be detected by our technique. We give arguments for the soundness and completeness of the checker generator, based on the automata process of Boulé and Zilic, using the formal semantics of PSL. We estimate the overhead of employing a full checker suite in a modern general-purpose processor, and we examine the algorithmic complexity of each sub-section of the checker-generator method. We identify cases which have the potential to require exponential space and time, and therefore might not be practical for implementing by our method.

B. CONTRIBUTIONS

The contributions of this research investigation are:

- A summary analysis of the processor malicious inclusion examples published to date.
- A novel process for formalizing security requirements, in processor designs, which derive from the behavioral requirements stated in an architectural specification. We are not aware of any other hardware security method by which the security of a particular processor *implementation* is specified in terms of a set of *architectural* requirements, which are expressed in a way that allows them to be dynamically evaluated in the implementation.
- A new method for dynamically enforcing processor security requirements that is effective across nearly all phases of design and implementation, from high-level design all the way to fielded operation. We are not aware of any other hardware security method by which the same stated behavioral requirements for a processor are enforceable in simulation, in FPGA emulation, and in fabricated processor samples.
- A technique for using assertion-checkers and code coverage simultaneously, in a complementary manner, to search for malicious inclusions during high-level simulation. Previous techniques used checkers or coverage in isolation.

- A demonstration, in a real general-purpose processor design, of how the method can be used to detect some, although not all, malicious inclusions–specifically, those which manifest as a violation of behavioral restrictions in the architectural specification.

- Creation of the most complete public-domain software tool for generating synthesizable runtime enforcement mechanisms in hardware, based on temporal logic specifications. The other publicly available checker generator, *synpsl*, covers only a portion of the PSL Simple Subset, outputs only VHDL, does not provide PSL abstract syntax trees, and does not implement DFA minimization [4].[25] The two most advanced checker generators described in the literature are FoCs and MBAC, which are not publicly available in source code [5], [6]. Our checker generator is public domain, covers the PSL Simple Subset, outputs VHDL or Verilog, provides PSL abstract syntax trees, and implements full DFA minimization, as well as some boolean simplifications that do not appear to be implemented in the other tools (See Table 13).

- A description of the algorithmic complexity for each step in the checker-generator method.

- A detailed analysis of the soundness and completeness of the automata-based checker-generator method, with respect to the PSL formal semantics.

C. RECENT RELATED WORK

In addition to the physical analysis and design analysis methods for MI detection, reviewed in Chapter V, several researchers have recently independently proposed methods, similar to ours, which employ assertions, or assertion-like formulas, in the context of hardware security.

Love, Jin, and Makris proposed a method for analyzing security properties of a design at the HDL level by translating a subset of Verilog into a specialized reasoning language called Coq [102]. They used automated analysis tools to prove whether various security properties, also specified in Coq, will hold. The desired security properties are specified along with the design, which in their example is a pair of register files, plus a module that can copy data between the files. The Coq theorems which express the desired security properties are very similar in appearance and function to assertions, like those in PSL and SVA. The hardware module in their demonstration is very small compared to a general-purpose processor, their method applies only in simulation, there is no methodology given for formulating assertions, and it is not clear how well full HDLs might be translated into the domain-specific proof language.

Zhang and Tehranipoor have proposed a method that, like ours, characterizes malicious or non-malicious behavior using assertions, and attempts to focus the search for malicious circuits through the aid of coverage techniques [112]. They used SystemVerilog Assertions (SVA) in their demonstration. However, they did not propose a structured method for developing the assertions, they used only a small hardware design in their demonstration, and their method applies only in simulation.

The independent use, by other researchers, of assertions in a security context for hardware designs increases our confidence in the application's potential value.

[25] We obtained the source code for this tool from the author, via a Creative Commons license.

D. RECOMMENDATIONS FOR FUTURE WORK

1. Analysis of Hardware Designs

Several specific subjects suggest themselves for future research. The first few are related to our method, and the last one is independent.

a. Related Tasks

One possible area of research is the validity of composition of multiple behavioral requirements, as specified in PSL assertions. It may be the case that two assertions contradict each other, so that a behavior may be permitted by one and prohibited by another. This will be important if we allow policy requirements to be specified both positively and negatively. If policy requirements are only specified negatively (i.e., the blacklist approach), and all non-prohibited behaviors are permitted, then no conflicts arise. However, any combination of whitelist and blacklist specifications (Behavior A is explicitly allowed, behavior B is explicitly prohibited) raises the possibility of conflict between requirements. This problem arises in access control policies, for example [113], and has been widely studied in that context. If temporal logic-based restrictions are expressed in hardware, some policy resolution strategy may be useful.

We believe it would be useful to perform adversarial experiments, where the group attempting to detect malicious inclusions is not involved with their development. This type of experiment has been done at the Embedded Systems Challenge during Cyber Security Awareness Week; however, the designs involved are much smaller than a full general-purpose processor [17]. In other demonstrations of MI detection to date (like ours), which do use a full-scale processor design, the experiments have been more academic in nature.

One portion of the assertion-checker method that can use improvement is the process of *mapping*, where the text-based behavioral requirements are mapped, from the architectural documents, to assertions that are specific to an implementation. This process requires the ability to identify requirements stated in a text, remove ambiguity from them, then map them to signals in one or more design units. Since the process is currently time-consuming, any automation or semi-automation would be useful.

Similarly, it would be useful to develop a method for automatically verifying the hierarchical completeness of the checkers. For example, if we implement a checker for signals *x, y*, and *z* in module A, and signals *x, y*, and *z* (or functionally equivalent copies of them) are present in subordinate module B (B is a sub-unit of A), then we need a checker instance for B, as well. It should be possible to automate this type of check.

We also recommend experiments involving commercial, rather than open-source, processor designs. An academic-corporate partnership might facilitate these, since the detailed design of modern commercial processors is not normally available to academia. Such experiments would help establish what processor applications are most suitable for application of the method.

Finally, we believe it would be useful to use an assertion-based method for security, like ours, *in conjunction* with the design of a processor throughout its entire development lifecycle, from inception to completion, rather than adding the security portion *after* the processor's high-level design is complete. This

type of approach is also endorsed by Love, Jin, and Makris [102]. This "design for security" approach would be similar to the "design for test" philosophy that is currently gaining popularity [44].

b. Independent Task

An unrelated investigation which interests us is the further study of signal dependency in hardware designs, across many design modules. Dependency analysis has long been used in the study of software, but less so in hardware. A technique called "program slicing" helps identify an entity's forward and backward dependencies, i.e., those units whose value depends on it, and those units upon which it depends. Clarke et al. introduced the idea of extending this process from sequential languages, like most software, to concurrent languages, like VHDL [114]. Vasudevan, Emerson, and Abraham demonstrated the use of HDL program slicing for improving functional verification [115] in hardware; we recommend exploration of whether some of these techniques can be useful in the security analysis of high-level hardware designs.

2. A General View

In the broader context, we make the following general observations and assessments regarding the future of hardware design security:

- The security analysis of hardware designs in the future will come to rely more on the use of assertions. We imagine that, rather than developing a design and adding assertions at the end, as in our demonstration, designers will develop hardware modules simultaneously with behavioral assertions right from the start. This sentiment is echoed by Foster, Krolnik and Lacey: "The way design and verification has traditionally been performed is changing. In the future, we predict that design and verification will become property-based" [116]. We also imagine assertions being integrated more often in hierarchical construction; when smaller, component modules are combined to form larger hardware designs, the assertions governing the smaller components will be integrated into assertions which cover the larger components, along the lines of the OVM Methodology [44].

- At the behest of customers with high-assurance applications, it will be desirable for some general purpose processors to be designed *from inception* with formally evaluatable security in mind, even at the cost of reduced features and performance, through simplified design. It would be useful in some of these applications to not only define a processor's behavioral requirements in detail up front, but to be able to automatically verify that they remain in force at every phase of the design lifecycle.

- There will likely be increased interest in hardware-software co-verification, where the focus of effort is on the application binary interface, or ABI. Formally verifying properties of hardware and software together adds complexity, but can also add to our confidence in the fidelity of the combined end product.

- There exists a need for verifying the correctness and security of hardware design tools, which are very complex collections of software. Our ability to perform this verification today is limited by the proprietary nature of commercial tools, as well as the lack of universal standards in many aspects of hardware design. Perhaps the development of commercial-quality open-source design tools will allow developers and customers to scrutinize and verify them more readily.

Because commercial general-purpose processors are designed for the mass market based on performance and cost, we do not expect that such processors will undergo the level of security analysis that high-assurance customers desire. As a result, we observe an increase in custom-designed processors and systems-on-chip for consumers like DoD, in a variety of applications. This divergence between the mass-market, high-performance, cost-based processors, fabricated overseas, and the domestically-produced, special-purpose, high-assurance processors, is likely to grow more pronounced over time. We believe that the principles outlined in this research can assist in evaluating the security of the latter group, especially, though they could be applied commercially, as well.

LIST OF REFERENCES

[1] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, "Trojan detection using IC fingerprinting," in *Security and Privacy, IEEE Symposium on*, pp. 296–310, May 2007.

[2] C. Sturton, M. Hicks, D. Wagner, and S. King, "Defeating UCI: Building stealthy and malicious hardware," in *Proceedings of the 32nd IEEE Symposium on Security and Privacy*, (Oakland, CA), May 2011.

[3] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design and Test of Computers*, vol. 27, no. 1, pp. 10–25, 2010.

[4] R. Findenig, "Behavioral synthesis of PSL assertions." M.S. thesis. Upper Austrian University of Applied Sciences, 2007.

[5] Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfsthal, "FoCs - automatic generation of simulation checkers from formal specifications," in *12th International Conference on Computer Aided Verification*, LNCS 1855, Springer, July 2000.

[6] M. Boule and Z. Zilic, *Generating Hardware Assertion Checkers*. Montreal, Canada: Springer, 2008.

[7] B. Sharkey, "DARPA TRUST in Integrated Circuits Industry Day Brief," March 2007.

[8] S. Adee, "The hunt for the kill switch," *Spectrum, IEEE*, vol. 45, pp. 34–39, May 2008.

[9] J. Roy, F. Koushanfar, and I. Markov, "Extended abstract: circuit CAD tools as a security threat," in *IEEE International Workshop on Hardware-Oriented Security and Trust*, pp. 65–66, June 2008.

[10] Office of the Undersecretary of Defense for Acquisition, Technology, and Logistics, "Report of the Defense Science Board task force on high performance microchip supply," tech. rep., February, 2005.

[11] R. McCormack, "DoD broadens 'trusted' foundry program to include microelectronics supply chain," *Manufacturing and Technology News*, vol. 15, pp. 1–5, February 2008.

[12] F. Yinug, "Challenges to foreign investment in high-tech semiconductor production in china," in *Journal of International Trade and Economics*, U.S. International Trade Commission, May 2009.

[13] D. Nystedt, "Intel Got Its New China Fab for a Bargain, Analyst Says," *CIO.com*, February 2010.

[14] S. Johnson, "Fake Chips Threaten Military," *San Jose Mercury News*, September 2010.

[15] J. Markoff, "Old Trick Threatens Newest Weapons," *New York Times*, October 2009.

[16] Woodmann, "AMD processors undocumented debugging features and MSRs." http://www.woodmann.com/forum/archive/index.php/t-13891.html, December 2010.

[17] Y. Jin, N. Kupp, and Y. Makris, "Experiences in hardware trojan design and implementation," in *Hardware-Oriented Security and Trust, IEEE International Workshop on*, pp. 50–57, 2009.

[18] S. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, "Designing and implementing malicious hardware," in *Proceedings of the 1st USENIX Workshop on Large-Scale Exploits and Emergent Threats*, pp. 1–8, USENIX Association, 2008.

[19] A. Waksman and S. Sethumadhavan, "Tamper evident microprocessors," in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, pp. 173–188, May 2010.

[20] X. Wang, M. Tehranipoor, and J. Plusquellic, "Detecting malicious inclusions in secure hardware: Challenges and solutions," in *Hardware-Oriented Security and Trust, IEEE International Workshop on*, pp. 15–19, 2008.

[21] J. Rajendran, E. Gavas, J. Jimenez, V. Padman, and R. Karri, "Towards a comprehensive and systematic classification of hardware trojans," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1871–1874, October 2010.

[22] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor, "Trustworthy hardware: identifying and classifying Hardware Trojans," *Computer*, vol. 43, pp. 39–46, October 2010.

[23] R. Rad, J. Plusquellic, and M. Tehranipoor, "Sensitivity analysis to Hardware Trojans using power supply transient signals," in *IEEE International Workshop on Hardware-Oriented Security and Trust*, pp. 3–7, June 2008.

[24] D. Song et al., "Bitblaze: A new approach to computer security via binary analysis," in *Proceedings of the 4th International Conference on Information Systems Security*, 2008.

[25] R. Hamming, *The Art of Doing Science and Engineering: Learning to Learn*. Amsterdam, The Netherlands: Gordon and Breach, 1997.

[26] M. Harrison, W. Ruzzo, and J. Ullman, "Protection in operating systems," *Communications of the ACM*, vol. 19, no. 8, pp. 461–471, 1976.

[27] D. Denning, "A lattice model of information flow," *Communications of the ACM*, vol. 19, pp. 236–243, May 1976.

[28] J. Goguen and J. Meseguer, "Security policies and security models," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 11–20, IEEE Computer Society Press, April 1982.

[29] D. Clark and D. Wilson, "A comparison of commercial and military security policies," in *Proceedings of the 1987 IEEE Symposium on Research on Security and Privacy*, 1987.

[30] J. Anderson, "Computer security technology planning study," Tech. Rep. ESD-TR-73-51, Electronic Systems Division, Air Force Systems Command, Hanscom AFB, Bedford, MA, October 1972.

[31] R. Kemmerer, "Shared resource matrix methodology: An approach to identifying storage and timing channels," *ACM Transactions on Computing Systems*, vol. 1, no. 3, pp. 256–277, 1983.

[32] Y. Zhou, P. Zhou, F. Qin, W. Liu, and J. Torrellas, "Efficient and flexible architectural support for dynamic monitoring," *ACM Transactions on Architectural Support for Code Optimization*, vol. 2, no. 1, pp. 3–33, 2005.

[33] G. Suh, D. Clarke, B. Gasend, M. van Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *36th Annual IEEE-ACM International Symposium on Microarchitecture*, pp. 339–350, 2003.

[34] F. Schneider, "Enforceable security policies," *ACM Transactions on Information System Security*, vol. 3, no. 1, pp. 30–50, 2000.

[35] M. Gordon, "PSL semantics in higher order logic," in *Proceedings of the 5th International Workshop on Designing Correct Circuits*, 2004.

[36] A. Slobodova, J. Davis, S. Swords, and W. Hunt, "A flexible formal verification framework for industrial scale validation," in *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pp. 89–97, July 2011.

[37] B. Alpern and F. Schneider, "Verifying temporal properties without temporal logic," *ACM Transactions on Programming Language Systems*, vol. 11, pp. 147–167, 1989.

[38] D. Geist, A. Landver, and B. Singer, "Formal verification of a processor's bus interface unit," tech. rep., August 1996.

[39] A. Goel and W. Lee, "Formal verification of an IBM CoreConnect processor local bus arbiter core," in *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, (New York, NY), pp. 196–200, ACM, 2000.

[40] A. Parash, "Formal verification of an MPEG decoder chip - a case study in the industrial use of formal methods," in *Proceedings of the Workshop on Advances in Verification (WAVe)*, 2000.

[41] C. Chavet, "Modelisation and validation of a chip embedded architecture for secure applications." M.S. thesis. TIMA Laboratory, Universite Joseph Fourier. Grenoble, France. 2003.

[42] V. Patankar, A. Jain, and R. Bryant, "Formal verification of an ARM processor," in *Proceedings of the Twelfth International Conference On VLSI Design*, pp. 282–287, 1999.

[43] E. Clarke, "The birth of model checking," in *25 Years of Model Checking* (O. Grumberg and H. Veith, eds.), vol. 5000 of *Lecture Notes in Computer Science*, pp. 1–26, Springer Berlin / Heidelberg, 2008.

[44] S. Iman, *Step by Step Functional Verification with SystemVerilog and OVM*. San Francisco, CA: Hansen Brown Publishing Company, 2010.

[45] T. Tuerk, K. Schneider, and M. Gordon, "Model checking PSL using HOL and SMV," in *Proceedings of the 2nd international Haifa verification conference on Hardware and software, verification and testing*, HVC'06, pp. 1–15, Springer-Verlag, 2007.

[46] A. Pnueli and A. Zaks, "PSL model checking and run-time verification via testers," in *Lecture Notes on Computer Science*, pp. 573–586, Springer, 2006.

[47] M. Tehranipoor and B. Sunar, "Towards hardware-intrinsic security: Foundations and practice," in *Information Security and Cryptography Texts and Monographs* (A. Sadeghi and D. Naccache, ed.), pp. 167–186, Springer, 2010.

[48] Defense Procurement News, "Defense Advanced Research Projects Agency, Arlington, VA, Contract Number HR0011-08-C0005," February 2010.

[49] Y. Alkabani and F. Koushanfar, "Consistency-based characterization for IC trojan detection," in *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '09, pp. 123–127, ACM, 2009.

[50] M. Banga and M. Hsiao, "Trusted RTL: Trojan detection methodology in pre-silicon designs," in *IEEE International Symposium on Hardware-Oriented Security and Trust*, (Anaheim, CA), pp. 56–59, June 2010.

[51] DARPA Microsystems Technology Office, "Broad Agency Announcement - Integrity and Reliability of Integrated Circuits," September 2010.

[52] M. Hicks, M. Finnicum, S. King, M. Martin, and J. Smith, "Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically," in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, pp. 159–172, May 2010.

[53] S. Adee, "IEEE spectrum tech talk blog: Trust in integrated circuits." http://spectrum.ieee.org/tech-talk/semiconductors/devices/trust_in_integrated_circuits, May 2008.

[54] The Common Criteria Portal. http://www.commoncriteriaportal.org/, August 2011.

[55] C. Eisner and D. Fisman, *A Practical Introduction to PSL*. New york, NY: Springer, 2006.

[56] IEEE, "Standard 1850-2010, for the Property Specification Language (PSL)," pp. 1–171, June 2010.

[57] Jones, C. B., "The early search for tractable ways of reasoning about programs," *Annals of the History of Computing, IEEE*, vol. 25, pp. 26–49, April-June 2003.

[58] A. Pnueli, "The temporal logic of programs," in *Foundations of Computer Science, 18th Annual Symposium On*, pp. 46–57, 1977.

[59] E. Clarke and E. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Proceedings of the Workshop on Logic of Programs, Lecture Notes in Computer Science*, 1981.

[60] C. Eisner, "PSL for runtime verification: Theory and practice," in *7th International Workshop on Runtime Verification*, pp. 1–8, March 2007.

[61] IEEE, "Standard 1800-2009, for the SystemVerilog Unified Hardware Design, Specification, and Verification Language," pp. C1–1285, 2009.

[62] IEEE, "Standard 1850-2005, for the Property Specification Language (PSL)," pp. 1–156, September 2005.

[63] B. Alpern and F. Schneider, "Recognizing safety and liveness," Tech. Rep. TR 86-727, Department of Computer Science, Cornell University, January 1986.

[64] IEEE, "Standard 1076-2008 (Revision of IEEE Std 1076-2002), for the VHDL Language Reference," 2009.

[65] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin, "Security checkers: Detecting processor malicious inclusions at runtime," in *IEEE International Symposium on Hardware-Oriented Security and Trust*, pp. 34–39, June 2011.

[66] MIPS Technologies. http://www.mips.com/, August 2011.

[67] R. Ford, "The wrong stuff?," *Security Privacy, IEEE*, vol. 2, pp. 86–89, May-June 2004.

[68] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *23rd Annual Computer Security Applications Conference (ACSAC)*, 2007.

[69] S. Vasudevan, *Effective Functional Verification: Principles and Processes*. Dordrecht, The Netherlands: Springer, 2006.

[70] M. Glinz, "On non-functional requirements," *IEEE International Conference on Requirements Engineering*, vol. 1, pp. 21–26, 2007.

[71] M. Boule and Z. Zilic, "Efficient automata-based assertion-checker synthesis of PSL properties," in *Eleventh Annual IEEE International High-Level Design Validation and Test Workshop*, pp. 69–76, IEEE Computer Society, November 2006.

[72] M. Boule and Z. Zilic, "Efficient automata-based assertion-checker synthesis of SEREs for hardware emulation," in *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, ASP-DAC '07, pp. 324–329, IEEE Computer Society, 2007.

[73] M. Vardi, "An automata-theoretic approach to linear temporal logic," in *Banff Higher Order Works. Lecture Notes on Computer Science*, pp. 238–266, Springer, 1996.

[74] P. Gastin and D. Oddoux, "Fast LTL to Buchi automata translation," in *Proceedings of the 13th International Conference on Computer Aided Verification*, pp. 53–65, Springer-Verlag, July 2001.

[75] J. Hopcroft, R. Motwani, and J. Ullman, *Introduction to Automata theory, Languages, and Computation*. Reading, MA: Pearson Addison-Wesley, 3 ed., 1979.

[76] L. Lamport, "Proving the correctness of multiprocess programs," *Software Engineering, IEEE Transactions on*, vol. SE-3, no. 2, pp. 125–143, 1977.

[77] T. Jiang and B. Ravikumar, "Minimal NFA problems are hard," *SIAM Journal on Computing*, vol. 22, pp. 1117–1141, December 1993.

[78] D. Baez, "PLY (Python Lex-Yacc) Homepage." http://www.dabeaz.com/ply/, August 2011.

[79] CompilerTools.net, "The Lex and Yacc Page." http://dinosaur.compilertools.net/, August 2011.

[80] Graphviz Organization, "Graphviz." http://www.graphviz.org/, August 2011.

[81] Mentor Graphics Corp., "QuestaSim user's manual, software version 10.0a.," 2011.

[82] A. Datta, J. Franklin, D. Garg, L. Jia, and D. Kaynar, "On adversary models and compositional security," *Security and Privacy, IEEE*, vol. 9, pp. 26–32, May-June 2011.

[83] M. Abramovici and P. Bradley, "Integrated circuit security: New threats and solutions," in *CSIIRW '09: Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research*, pp. 1–3, ACM, 2009.

[84] G. Loh, Y. Xie, and B. Black, "Processor design in 3D die-stacking technologies," *IEEE Micro*, vol. 27, no. 3, pp. 31–48, 2007.

[85] S. Lim, "TSV-based 3D-IC research activities at the Georgia Tech Computer-Aided Design Laboratory." August 2010.

[86] ITRS, "International Technology Roadmap for Semiconductors." http://www.itrs.net/reports.html, 2007.

[87] E. Beyne et al., "Through-silicon via and die stacking technologies for microsystems integration," in *IEEE International Electronic Devices Meeting*, pp. 1–4, 2008.

[88] P. Emma and E. Kursun, "Opportunities and challenges for 3D systems and their design," *Design and Test of Computers, IEEE*, vol. 26, pp. 6–14, September-October 2009.

[89] S. Mysore, B. Agrawal, N. Srivastava, S. Lin, K. Banerjee, and T. Sherwood, "Introspective 3D chips," in *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 264–273, ACM, 2006.

[90] T. Huffmire, T. Levin, M. Bilzor, C. Irvine, J. Valamehr, M. Tiwari, T. Sherwood and R. Kastner, "Hardware trust implications of 3-D integration," in *6th Workshop on Embedded Systems Security*, 2010.

[91] M. Bilzor, "3D execution monitor: Using 3D circuits to detect hardware malicious inclusions in general purpose processors," in *International Conference on Information Warfare*, March 2011.

[92] J. Rushby, "Kernels for safety?," in *Safe and Secure Computing Systems*, ch. 13, pp. 210–220, Blackwell Scientific Publications, 1989. (Proceedings of a Symposium held in Glasgow, October 1986).

[93] B. Alpern and F. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21, pp. 181–185, October 1985.

[94] C. Eisner. Personal communication.

[95] V. Gligor, "A note on denial-of-service in operating systems," *IEEE Transactions on Software Engineering*, vol. SE-10, pp. 320–324, May 1984.

[96] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri, "How low can you go?: recommendations for hardware-supported minimal TCB code execution," *SIGARCH Computer Architecture News*, vol. 36, pp. 14–25, March 2008.

[97] OpenCores Foundation. http://opencores.org/, August 2011.

[98] R. Fajardo, "Minimal OpenRISC system on chip user manual. OpenCores.org," September 2010.

[99] Ubuntu Organization. http://www.ubuntu.com/, August 2011.

[100] GNU Software Foundation. http://www.gnu.org/, August 2011.

[101] OpenCores, "OpenRISC 1000 architecture manual," April 2006.

[102] E. Love, Y. Jin, and Y. Makris, "Enhancing security via provably trustworthy hardware intellectual property," in *IEEE International Symposium on Hardware-Oriented Security and Trust*, pp. 12–17, June 2011.

[103] K. Morin-Allory, M. Boulé, D. Borrione, and Z. Zilic, "Validating assertion language rewrite rules and semantics with automated theorem provers," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, pp. 1436–1448, September 2010.

[104] K. Schneider, *Verification of reactive systems: formal methods and algorithms*. Texts in theoretical computer science, Berlin, Germany: Springer, 2004.

[105] D. Bustan, D. Fisman, and J. Havlicek, "Automata construction for PSL, technical report MCS05-04," tech. rep., The Weizmann Institute of Science. Haifa, Israel, 2005.

[106] A. Cimatti, M. Roveri, S. Semprini, and S. Tonetta, "From PSL to NBA: a modular symbolic encoding," in *Formal Methods in Computer Aided Design, FMCAD '06*, pp. 125–133, November 2006.

[107] S. Ben-David, R. Bloem, D. Fisman, A. Griesmayer, I. Pill, and S. Ruah, "Automata construction algorithms optimized for PSL," 2005. Property-Based System Design (PROSYD) Deliverable 3.2/4.

[108] MIPS Technologies, Inc., "MIPS architecture for programmers, Vol. I-III," 2010.

[109] J. Stokes, "Two Billion-Transistor Beasts: POWER7 and Niagara 3," *Ars Technica*, February 2010.

[110] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. Cambridge, MA: The MIT Press, 2001.

[111] R. Chakraborty, F. Wolff, S. Paul, C. Papachristou, and S. Bhunia, "MERO: A statistical approach for Hardware Trojan detection," in *Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '09, pp. 396–410, Springer-Verlag, 2009.

[112] X. Zhang and M. Tehranipoor, "Case study: Detecting Hardware Trojans in third-party digital IP cores," in *IEEE International Symposium on Hardware-Oriented Security and Trust*, pp. 67–70, June 2011.

[113] M. Dekker, J. Crampton, and S. Etalle, "RBAC administration in distributed systems," in *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*, pp. 93–101, 2008.

[114] E. Clarke, M. Fujita, S. Rajan, T. Reps, S. Shankar, and T. Teitelbaum, "Program slicing for VHDL," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 4, pp. 125–137, October 2002.

[115] S. Vasudevan, E. Emerson, E. Allen, and J. Abraham, "Improved verification of hardware designs through antecedent conditioned slicing," *International Journal of Software Tools and Technology Transfer*, vol. 9, pp. 89–101, February 2007.

[116] H. Foster, A. Krolnik, and J. Lacey, *Assertion-Based Design*. New York, NY: Kluwer Academic Publishers, 2004.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Fort Belvoir, Virginia
2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA