AD Viences in De bug Automation for a Modern Verification Environment

by

Brian Keng

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2013 by Brian Keng
Abstract

Advances in Debug Automation for a Modern Verification Environment

Brian Keng
Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto
2013

Over the past three decades, the growing list of requirements for integrated circuits has continually presented new challenges to the electronic design community. One of the biggest challenges in the design process is that of functional debugging, which aims to find the root-cause of a functional failure after it has been detected. In recent years, this key challenge has grown in size and scope as bugs commonly appear in both the design and verification environment. This increase in size and scope has made functional debugging one of the largest bottlenecks in the design cycle and points to an urgent need for more scalable and innovative debugging solutions.

This dissertation presents multiple novel contributions that address the challenges of increased size and scope of modern functional debugging. In particular, these contributions address the scalability of existing automated design debugging techniques, as well as introduce novel automated tools specifically for debugging the verification environment.

The first contribution introduces an unsatisfiable core-guided abstraction and refinement technique for design debugging that focuses on managing the design size aspect of debugging complexity. The second contribution introduces a path-directed abstraction and refinement technique that aims to manage the error trace length aspect of debugging complexity. The third contribution presents a novel method that utilizes unsatisfiable cores in design debugging to manage the multiple design errors aspect of debugging complexity. The fourth contribution presents an automated technique to aid debugging of errors found within formal properties themselves. The final contribution presents an automated technique to aid debugging of missing assumptions that are needed during verification methodologies that use formal methods.
Acknowledgements

First and foremost I would like to sincerely thank my Ph.D. supervisor Professor Andreas Veneris for being an excellent mentor, guide and teacher throughout my journey into research. You have been a constant source of motivation and guidance every step of the way.

I would also like to thank my parents who are an infinite source of love and support. You have instilled in me the values and lessons that have guided me throughout my life. I am forever grateful. Many thanks to my brother and sister whose love and support always brings a smile to my face, and to my girlfriend Jane Wu for her never-ending encouragement in everything I do.

Thank you to my colleagues at the University of Toronto and Venna Technologies for all your shared wisdom and constructive feedback. Special thanks are due to Duncan Exon Smith, Sean Safarpour, Hratch Mangassarian, Evean Qin, Alan Baker, Terry Yang, Bao Le, Dipanjan Sengupta and Zissis Poulos.

I am also grateful to my Ph.D. committee members Professor Jason Anderson, Professor Vaughn Betz, Professor Stephen Brown, Professor Sharad Malik, Professor Konstantinos Plataniotis, and Professor Charles Rackoff for their excellent feedback and insightful suggestions for my dissertation.

Finally, acknowledgments are due to the Natural Sciences and Engineering Research Council of Canada (NSERC), the Queen Elizabeth II Graduate Scholarship in Science and Technology, and the University of Toronto for their financial support.
Contents

List of Tables ix
List of Figures xi
List of Algorithms xiii

1 Introduction 1

1.1 Motivation .......................................................... 1

1.1.1 Functional Verification and Debugging .................. 4

1.1.2 Current State of Debugging Tools ......................... 6

1.1.3 Debug Automation ............................................. 7

1.2 Contributions ....................................................... 9

1.2.1 Core-Guided Abstraction and Refinement ............... 10

1.2.2 Path Directed Abstraction and Refinement ............. 11

1.2.3 Debugging of Multiple Errors Using UNSAT Cores .... 11

1.2.4 Automated Debugging of Assertions ...................... 12

1.2.5 Automated Debugging of Missing Assumptions ......... 12

1.3 Thesis Outline ......................................................... 13

2 Background 14

2.1 Introduction .......................................................... 14

2.2 Functional Verification ............................................ 14

2.2.1 Iterative Logic Array ........................................ 17
## List of Tables

1.1 Types of Errors ................................................................. 6

2.1 CNF for Basic Logic Gates ................................................... 23

2.2 Common SystemVerilog Assertion Operators ........................... 37

3.1 Design Statistics .............................................................. 55

3.2 Abstraction and Refinement Experiments ............................... 60

4.1 Design Characteristics ...................................................... 87

4.2 Refinement Improvement Experiments ................................... 88

4.3 Window Size Experiments .................................................. 88

4.4 Flexible PathDebug Experiments ........................................ 95

4.5 Trace Length Experiments ................................................ 96

5.1 Instance Information ....................................................... 105

5.2 Debugging Multiple Design Errors Experiments ....................... 106

6.1 Boolean Expression Mutations ........................................... 114

6.2 Arithmetic, Logic and Bitwise Mutations .............................. 115

6.3 Sequential Binary Mutations ............................................. 115

6.4 Sequential Concatenation Mutations ................................... 116

6.5 Sequential Repetition Mutations ........................................ 116

6.6 Implication Mutations ..................................................... 117

6.7 Instance Data ................................................................. 120
List of Figures

1.1 Simplified IC Design flow ........................................ 2

2.1 Time-frame Expansion Example .................................. 18
2.2 Visualization of Cardinality Constraints ..................... 24
2.3 Resolution Graph .................................................. 26
2.4 Implication Graph with Conflict ................................. 30
2.5 SAT-based Design Debugging ..................................... 35
2.6 Timing Specification ............................................... 39

3.1 SAT-based Design Debugging Example ......................... 43
3.2 Suspect Guided Abstraction and Refinement Example .... 46
3.3 Example of Abstraction ............................................ 49
3.4 Example of Refinement ............................................. 51
3.5 Example of Sequential Refinement ............................. 54
3.6 Solutions for Multiple Refinement Iterations ............... 58

4.1 Time Diagnosis Example ........................................... 65
4.2 Path Directed Abstraction and Refinement .................... 69
4.3 High-level Example of Path Directed Abstraction and Refinement 70
4.4 Abstraction Example .............................................. 72
4.5 Refinement Example ................................................ 76
4.6 Relative Memory for Different Window Sizes ............... 90
4.7 Length of Error Trace Analyzed ................................ 93
**List of Algorithms**

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The DPLL algorithm</td>
<td>27</td>
</tr>
<tr>
<td>2.2</td>
<td>Modern SAT algorithm</td>
<td>31</td>
</tr>
<tr>
<td>3.1</td>
<td>Core-Guided Abstraction and Refinement</td>
<td>53</td>
</tr>
<tr>
<td>4.1</td>
<td>Counter-Example Guided Abstraction and Refinement</td>
<td>67</td>
</tr>
<tr>
<td>4.2</td>
<td>Path Directed Abstraction and Refinement</td>
<td>78</td>
</tr>
<tr>
<td>4.3</td>
<td>Solution Verification</td>
<td>79</td>
</tr>
<tr>
<td>4.4</td>
<td>Flexible Path Directed Abstraction and Refinement</td>
<td>84</td>
</tr>
<tr>
<td>5.1</td>
<td>Debugging Multiple Design Errors</td>
<td>103</td>
</tr>
<tr>
<td>7.1</td>
<td>Generating Multiple Counter-Examples</td>
<td>139</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Over the past three decades, from automobiles to smart phones, modern computing devices have become ubiquitous in every aspect of our lives. At the heart of each of these computing devices is a compact integrated circuit (IC) that contains billions of transistors performing billions of calculations per second, all within a semiconductor die no larger than a fingernail. With an ever increasing list of functional, performance, physical size and power requirements, a massively complex IC design flow has evolved over the years to ensure that these ubiquitous devices can meet the growing demands of their underlying systems.

The modern IC design flow begins with a high-level written outline of its specifications, and through a series of complex design steps, culminates with billions of transistors fabricated onto a single piece of semiconductor material. To achieve this great feat, the complexity of the problem is managed through multiple layers of abstractions that iteratively translate the initial design specifications into increasingly detailed levels of implementation.

Figure 1.1 shows a simplified view of the IC design flow, which begins with a behavioral specification of the chip. This can be a natural language document or more precisely refined with a high-level behavioral model in languages such as Matlab or C. From this behavioral specification, behavioral synthesis translates it into a register transfer level (RTL) description such as Verilog or VHDL. This translation is typically performed manually by an RTL
Chapter 1. Introduction

Design

- Behavioral Specifications
- RTL Description
- Logic Synthesis
- Gate-level Netlist
- Layout Synthesis
- Layout
- Fabrication
- Chip

Verification, Testing and Debug

- Functional Verification
- Equivalence Checking
- Testing
- Silicon Debug / Fault Diagnosis
- Design Debugging

Figure 1.1: Simplified IC Design flow
designer, however, tools that automate this process have recently grown in popularity \cite{25, 56}. After an RTL description is available, logic synthesis is performed which translates the RTL into a gate-level netlist composed of basic logic gates that are available in the desired technology. Due to the availability of mature and robust technology, this step is generally performed using automated tools. At this stage in the flow, the gate-level netlist is then translated to a physical layout corresponding to masks needed to manufacture the semiconductor chip. This step is typically performed using an automated place and route tool, but can be performed manually for specific components depending on performance, power and related requirements. Finally, once physical layout is complete, the design can be sent for fabrication to create the final chip.

Each one of the above design implementations is accompanied by an equivalent verification/test step to ensure that the translation from a higher level of abstraction matches the next more detailed one. The first step in this process is functional verification (also known as design verification, or simply verification) which compares the behavioral specification to the RTL description. This is a comprehensive process utilizing a wide range of techniques and tools to ensure that no functional differences exist between the two abstractions. After it is complete, the next step compares the RTL description against the synthesized gate-level netlist during equivalence checking. Similar to its design counterpart, it is typically performed using a mature automated tool. If a mismatch occurs between any of these two pairs of descriptions, design debugging occurs to determine the root-cause of the functional error so it can be fixed at these early stages of design. Finally, the last step in this process is testing which ensures that the final chip matches the gate-level netlist. If a mismatch if found during this step, either silicon debug or fault diagnosis occurs, which aims to detect either functional errors or manufacturing defects, respectively.

The process of translating high-level specifications to a physical chip with billions of transistors is highly error prone. A single error in the RTL description may propagate to the fabricated chip, resulting in the difference between a functional IC and a useless slab of silicon. This situation has the potential to jeopardize project time lines and introduce millions of dollars in additional non-recurring engineering cost. In an effort to avert this costly situation, modern flows have adopted more rigorous verification and testing methodologies which can contribute
up to 70% of the total chip design time \[117\]. Moreover in recent years, a disproportionate amount of this effort has been skewed towards functional verification due to the orders-of-magnitude increase in cost for finding functional bugs in silicon prototypes \[141\]. This trend is also supported by the fact that many errors do in fact escape functional verification and appear in silicon prototypes. This results in 60% of application specific integrated circuits (ASIC) to fail not because of timing or power issues, but because of functional errors \[73\]. It is then no surprise that functional verification can contribute up to 46% of the total design time \[59\].

One of the key steps contributing to the functional verification bottleneck is that of debugging, which is estimated to take up to 50% of the total verification time \[58\]. This is confirmed by a recent report published by the International Technology Roadmap for Semiconductors (ITRS), an organization sponsored by chip manufacturers from across the world, describing debugging as “manual analysis of very long and complex bug traces”, further stating that: “Without major breakthroughs, design verification will be a non-scalable, show-stopping barrier to further progress in the semiconductor industry.” \[71\] Undoubtedly, this roadmap acknowledges that the current trends in verification and debug cannot continue. It highlights several key areas including greater use of automation and higher levels of abstraction to increase efficiency for both verification and debug. All of these areas need to be addressed to ensure the success and growth of the semiconductor industry continues at the same pace it has for the past 30 years.

1.1.1 Functional Verification and Debugging

Modern designs are composed of an interwoven ecosystem of in-house, legacy and acquired IP design blocks \[71\]. These system-on-chips (SoC) stitch together many different functions, protocols and engineers, making the entire system increasingly opaque and thus difficult to verify and debug. *Functional verification* (also known as design verification, or simply verification) is a key process in the design of these systems ensuring that the RTL description conforms with its corresponding behavioral specifications. To support this process, thousands of lines of verification code must be written that never gets synthesized to the final chip. Instead, this code models the behavioral specifications and checks its functionality against the design
implementation. In this thesis, we will broadly denote this non-design code as the verification environment.

Modern verification environments have evolved, out of necessity, into a diverse collection of different technologies built to efficiently verify complex SoCs. Traditional verification techniques have been augmented with advanced languages, methodologies, and technologies to dramatically increase verification efficiency. In conjunction, new tools such as automated testbench generators, functional coverage tools, property checkers, and equivalence checkers have all proliferated into industrial verification flows to deal with the manifold challenges involved. Despite these new tools and technologies, the challenges of this task remain immense as demonstrated by the growing 3:1 ratio between the number of verification engineers and designers.

One of the biggest challenges underlying verification is that of functional debugging, which is estimated to consume up to half of the total verification time. This process aims to find the root-cause of a failure after it has been detected during verification. In the past 30 years, the dominant portion of this task concentrated on debugging errors in the design implementation (e.g., RTL description). This was caused in part by the relative simplicity of the verification environment, which consisted primarily of testbench code that simulated the design against a set of pre-generated test vectors. However, in recent years, the verification environment has become exceedingly complex with a heterogeneous mix of new tools and technologies. This evolution has produced massive amounts of verification code that rivals in size and complexity to the design itself. Naturally, this led to an enormous proportion of debugging time spent analyzing, not design errors, but rather errors in the verification environment. As a result, modern functional debugging has dramatically increased in size and scope as bugs commonly appear in both the design and verification environment.

Table categorizes the two different types of errors that occur during functional verification. Broadly speaking, errors are categorized as either design errors that are located in the RTL (or gate-level netlist), or verification environment errors located in the testbench, assertions, or other verification code. Design errors are typically analyzed by engineers who write RTL descriptions, while verification errors are handled by a separate set of verification engineers re-
Table 1.1: Types of Errors

<table>
<thead>
<tr>
<th>Bug-Type</th>
<th>Engineer to Debug</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design</td>
<td>Design Engineer</td>
<td>RTL/gate-level netlist</td>
</tr>
<tr>
<td>Verification Environment</td>
<td>Verification Engineer</td>
<td>Non-synthesizable testbench (SystemVerilog, SystemC, C, Matlab, etc.), assertions, environment</td>
</tr>
</tbody>
</table>

1.1.2 Current State of Debugging Tools

Current state-of-the-art industrial debugging tools provide some relief to the many challenges involved in functional debugging. These tools \[66, 130\] use graphical waveform viewers, code tracing, “what-if” analysis engines, and novel visualization techniques to allow both designer and verification engineer to intuitively navigate a failure. Although these solutions improve upon their predecessors, the underlying technology of these tools has been available for over a decade. Moreover, these tools were designed to primarily handle RTL debugging issues without significant consideration for the more recent verification environment bugs. Even when considering RTL bugs, trends such as design blocks reaching half a million synthesized gates outline the difficulty of scaling these tools for modern designs. These issues, along with their inherent manual nature, drastically limit their ability to efficiently handle the growing challenges of modern functional debugging.

These challenges all point to the urgent need for more innovative solutions in functional debugging. This necessitates the introduction of more debug automation to reduce the dependence on manual engineering effort, and solutions to handle the growing size of designs and the growing scope of the debug problem. Only when these two issues are addressed will the debugging burden be resolved and the critical bottleneck in the verification process be lifted.
1.1.3 Debug Automation

As discussed in the previous subsection, industrial solutions to debugging have not met the urgent need for more debug automation. These solutions have been predominantly incremental, based upon existing techniques (e.g., waveforms, code tracing etc.), in an effort to increase the efficiency an engineer manually analyzing a failure. However, these incremental advances have not delivered the orders-of-magnitude increase in productivity demanded by modern design and verification flows. Despite this lack of automation in industrial solutions, the academic community has been researching methods to automate various aspects of debugging over the past 20 years.

The earliest works on debugging were concentrated in fault diagnosis – the process of diagnosing manufacturing defects within a silicon prototype. These techniques [6, 7, 10] aim to detect defects by injecting a pre-defined fault into the netlist, simulating it, and comparing the result until it matches the observed silicon prototype. In contrast, design debugging (also known as design error diagnosis) methods aim to localize the root-cause of a logic or functional error (versus just manufacturing defects). Naturally, the earliest methods for design debugging were based upon the related fault diagnosis techniques [68, 135, 137], which were shown to be effective in localizing a wide range of logic errors [5]. Subsequently, Binary Decision Diagrams (BDD) [86] were used to a similar effect [69] for logic debugging.

More recently, symbolic design debugging techniques based upon Boolean Satisfiability (SAT) [124, 128], maximum SAT (max-sat) [31] and Quantified Boolean Formula (QBF) [95] have proliferated due to the orders-of-magnitude increase in capacity and performance over previous techniques. By adapting and borrowing many concepts from related verification problems in model checking [100, 144] and equivalence checking [61, 62], these debugging techniques have become the focus of state-of-the-art research in debug automation. These techniques not only address design debugging but have also been extended to automate various aspects of the silicon debug process [27, 45, 46, 142] as well.

Many extensions and improvements have been proposed on the basic design debugging formulations that aim to expand both its applicability and scalability. Research on extending automated design debugging techniques to work with RTL [11, 28] and in the presence of asser-
tions [55], have greatly increased its relevance in a modern design flow. In parallel, significant
effort has been spent to increase scalability [81, 82, 121, 131]. These techniques manage the
various aspects of design debugging complexity in an effort to handle industrial scale designs.

In general, design debugging complexity has three key parameters: design size, error trace
length, and number of simultaneous errors. The design size relates directly to the size a par-
ticular implementation (e.g., RTL or gate-level netlist). This has roughly been doubling every
18-24 months, as predicted by Moore’s Law [106]. The error trace length relates to the length
of time the error takes from excitation to observation. This length has grown considerably
longer alongside the enormous size of the sequential state space, reaching thousands of cycles
in a typical simulation run [26]. The number of simultaneous errors relates to the occurrence
of one or more errors directly causing a single failure. This too has become more likely as the
number of design components rise. The tremendous growth in these three parameters have
severely limited the ability of automated design debugging techniques to handle industrial scale
designs. To realize the full benefit of these techniques, further research in scaling debugging
solutions for industrial problems will be needed.

Meanwhile, the growth in the parameters of design debugging complexity has led to highly
complex verification environments necessary to ensure functional correctness. This recent trend
necessitates greater debug automation in order to ensure it does not become a bottleneck.
Unfortunately, automated design debugging techniques are poorly suited towards debugging
the verification environment for two main reasons. First, most verification environment code
is non-synthesizable, which excludes the majority of automated circuit debugging techniques.
Second, circuit debugging techniques predominantly rely on localization of the error to aid
debugging. However, this proves ineffective in many types of synthesizable verification code
(e.g., assertions) which are written in higher-levels of abstraction consisting of fewer but more
complex lines of code. This shifts the bulk of the verification environment debugging effort
from answering the question “where is the bug?” to “why is the bug occurring?” As a result,
the advances in automated design debugging techniques provide little help when debugging the
verification environment. This leads to a gaping hole in the verification flow where automation
is urgently needed.
To ensure that neither of these complementary debugging scenarios becomes the “show-stopping barrier to further progress in the semiconductor industry” [71], new solutions must be found to address these issues. For design debugging, novel and robust techniques to scale existing automated design frameworks for industrial problems must be discovered so that the burden can be lifted off the design engineer. Concurrently for the verification environment, novel methods for debug automation are needed to solve the issues that have arisen out of the recent trends of massive verification environments. Addressing the above issues is critical to relieving the functional debugging pain and the continued success of the semiconductor industry.

1.2 Contributions

This thesis aims to alleviate the debugging burden by addressing several key issues in the functional debugging process. The first core area of contribution is in the scalability of existing automated design debugging techniques. In particular, this thesis introduces three novel techniques to address the three key parameters of debugging design complexity: design size, error trace length and number of simultaneous errors. In summary, these contributions are as follows.

- The first contribution [83] introduces a core-guided abstraction and refinement technique for design debugging that focuses on managing the design size aspect of debugging complexity. It improves upon past techniques by providing a finer grain refinement algorithm, as well as stronger guarantees with respect to completeness of the method.

- The second contribution [85] introduces a path-directed abstraction and refinement technique that aims to manage the error trace length aspect of debugging complexity. The technique is able to vastly reduce the debugging problem size by iteratively analyzing fixed size windows of the error trace.

- The third contribution [78] presents a novel method that utilizes unsatisfiable (UNSAT) cores in design debugging to manage the multiple design errors aspect of debugging complexity. It generalizes previous work by providing a method to both generate and utilize UNSAT cores, allowing it to be more widely applicable.
The second core area of contribution, to the best of our knowledge, is the first attempt in the literature to introduce automated tools specifically for debugging the verification environment. In particular, the contributions address two important debugging problems related to assertion-based verification. In summary, the contributions in this area are as follows.

- The first contribution\cite{79, 80} presents an automated technique to aid debugging of errors found within formal properties themselves. This aims to tackle a class of errors in the verification environment specifically dealing with assertions.

- The second contribution\cite{84} presents an automated technique to aid debugging of missing assumptions that are needed during verification methodologies that use formal methods. This also tackles an important class of debug problems arising from the need to specify assumptions to model the surrounding environment during formal verification.

In the following subsections, we will discuss each of the contributions in greater detail.

### 1.2.1 Core-Guided Abstraction and Refinement

Abstraction and refinement is a general methodology in model checking and design debugging for solving complex instances. The basic idea behind this family of techniques is to generate an abstraction which represents a simplified approximation of the original concrete problem. Ideally, this abstraction is a strong enough approximation to perform the desired analysis.

However, frequently this is not the case, and the abstraction must be refined into a stronger approximation with respect to the original problem. After a sufficient number of refinement iterations, the model contains enough of the original problem such that a complete analysis can be performed. This general concept can be realized in many ways by defining different types of both abstraction and refinement techniques.

This contribution introduces a novel abstraction and refinement algorithm for design debugging that improves upon previous work. The initial abstraction removes all components of the design leaving only the primary input/outputs. It then iteratively adds back logic within hierarchical modules guided by the use of UNSAT cores, addressing primarily the design size aspect of debugging complexity. Unlike previous work, the abstraction at each stage is proven to
represent a strict under-approximation of the concrete debugging problem, guaranteeing sound and complete results for the overall algorithm.

1.2.2 Path Directed Abstraction and Refinement

This contribution presents a novel design debugging algorithm utilizing time-windows in an abstraction and refinement framework addressing the error trace length aspect of debugging complexity. The process begins by dividing the error trace into non-overlapping time-windows where each window is analyzed separately. Like other window-based methods, only one time-window is explicitly modeled while subsequent time-frames are over-approximated using the abstraction. The key novelty in this work is the use of path-based abstraction to model parts of the error trace that directly lead to the observed failure during simulation. This is accomplished by modeling only those paths in the time-frame expanded circuit that directly affect the observed failure. Additionally, unlike previous work, it provides an efficient method to refine this abstraction using an iterative technique based upon the conflict graph returned by the SAT-solver. The theory behind this abstraction and refinement method is proven to be sound and complete. The net result is a dramatically reduced memory footprint that mitigates the incompleteness issues of past time-windowing methods.

1.2.3 Debugging of Multiple Errors Using UNSAT Cores

In the context of design debugging, an unsatisfiable (UNSAT) core intuitively represents a group of paths in the time-frame expanded circuit from which the primary inputs and initial states propagate to the expected primary outputs and result in a conflict. This contribution utilizes this observation and presents a novel algorithm to deal with debugging multiple design errors using UNSAT cores.

This contribution generalizes previous work by showing how to both generate and apply unsatisfiable cores for multiple design errors without the need for overlapping UNSAT cores. The generation of the UNSAT cores comes as a by-product of solving the SAT-based debugging formulation at each error cardinality. By using the cores generated at each cardinality, many locations can be ruled out that are provably not part of solutions at higher error cardinalities.
This effectively reduces the exponential solution space for multiple simultaneous design errors.

### 1.2.4 Automated Debugging of Assertions

The adoption of temporal assertions in modern verification environments introduces new challenges to the debugging process. Temporal assertions concisely define sequential behaviors across multiple cycles and execution threads, creating a significant paradigm shift from RTL. As a result, modern temporal assertion languages such as SystemVerilog assertions and Property Specification Language (PSL) are foreign to most engineers who are more familiar with RTL semantics rather than these concisely specified formal properties. For these reasons, bugs frequently occur with complex temporal assertions and remain one of the biggest challenges in their widespread adoption.

This contribution presents a novel automated debugging methodology for modern assertion languages that takes a different approach compared to traditional RTL debug automation methods. Rather than localizing the error, it aids debugging by generating a set of properties closely related to the original failing assertion that have been validated against the RTL. These properties serve as a basis for possible corrections to the failing assertion, providing an intuitive method for debugging and correction. They also provide insight into design behavior by being able to contrast their differences with the failing assertion. The result is increased debugging efficiency by removing much of the manual guesswork needed when debugging assertions.

### 1.2.5 Automated Debugging of Missing Assumptions

Formal tools increase functional verification efficiency by exhaustively searching a design’s state space for hard to find bugs. Often the counter-examples returned from these tools are not due to design bugs but missing assumptions that are needed to model the surrounding environment. These types of false counter-examples have become a great concern in the industry today. To address this issue, input assumptions are typically added by the engineer to restrict the input space a formal tool is allowed to explore. These assumptions are difficult to generate as they are usually implicit in the documentation or implementation of adjacent design blocks. As a consequence, this process reduces the efficiency of formal methodologies because missing input
assumptions must be determined before real design bugs can actually be detected.

In this contribution, we present an algorithm to automatically generate missing input assumptions given a failing counter-example. It begins by generating multiple formal counter-examples for the error. This is accomplished by iteratively extracting a minimal set of failing input behaviors that led directly to the failure observed in the counter-example. Next, a logic function is constructed that encodes these bad input combinations that caused the failure. Using this function, small SAT instances are created to filter out a dictionary of fixed cycle properties based on the input signals. The net result returned to the user is a list of input assumptions that restrict the bad behaviors seen in the counter-examples. This list can either contain the actual missing assumption, or may give strong intuition about which assumption is needed, greatly simplifying the tedious process of manually debugging these failures.

1.3 Thesis Outline

This thesis is organized as follows. Chapter 2 provides necessary background material for the contributions. Chapter 3 presents the core-guided abstraction and refinement technique, while Chapter 4 presents the path directed abstraction and refinement technique. The debugging of multiple errors using UNSAT cores is presented in Chapter 5. Chapter 6 and Chapter 7 present the techniques to debug errors within assertions and missing assumptions, respectively. Finally, Chapter 8 concludes this thesis and discusses future research directions.
Chapter 2

Background

2.1 Introduction

This chapter presents background material that is relevant to the contributions in this dissertation. Section 2.2 and Section 2.3 define and discuss terms related to functional verification and debugging, respectively. Background material on Boolean Satisfiability (SAT) is described in Section 2.4, followed by the basics of SAT solvers in Section 2.5. Section 2.6 summarizes the basic SAT-based design debugging formulation, while Section 2.7 introduces concepts relating to formal properties and SystemVerilog Assertions. Finally, Section 2.8 summarizes this chapter.

2.2 Functional Verification

Functional verification (also known as design verification, or simply verification) is the step in the design flow that ensures that a design implementation (e.g., RTL description) conforms with its corresponding behavioral specification. Although many types of non-functional verification exist, such as power or timing, only functional verification will be discussed in this thesis. Similarly, many of the terms defined in this section may have alternate meanings in other parts of the design flow; however, unless otherwise specified, they will only be used in the context as defined below.
**Definition 1** A failure is a discrepancy between a design implementation (e.g., RTL) and its specification at one or more observation points (e.g., primary output pins, assertions) under the same initial conditions and primary input stimulus.

Broadly speaking, the aim of functional verification is to identify and fix as many failures as possible before moving onto the next, more detailed, design abstraction. A discrepancy or mismatch observed in a failure takes the form of conflicting Boolean values at any observation point in the design. Observation points typically correspond to primary output pins, but can also be thought of as internal circuit lines or failing assertions. These latter two cases can be transformed into the former by simply considering the associated circuit line as a primary output pin. In the case of an assertion, this requires one to synthesize the assertion and make its output signal a primary output pin.

**Definition 2** An error, or bug, is a component or structure within the design implementation (e.g., RTL) or specification (e.g., testbench, assertions) that produces a functional mismatch between the two representations causing a failure.

Failures are produced by errors that can exist in both the design implementation or behavioral specification. The former case is known as a design error and can be caused by a change in the specifications, software bugs in the automated tools, or the human factor. The latter case is known as a verification error (sometimes colloquially described as a testbench bug or error), and can be caused by incorrect translations of the behavioral specifications into verification code such as testbenches or assertions.

After a failure is detected during verification, an error trace or counter-example can usually be generated describing the conditions in which the failure can be reproduced. The error trace contains precisely the information needed to excite and observe the erroneous behavior. By simulating the erroneous design with the error trace, one can observe a mismatch between the design implementation and behavioral specification.

**Definition 3** An error trace ($\mathcal{V}_0^k$) of length $k + 1$ for clock-cycles $0$ to $k$ consists of three components:
• An initial state predicate \((S^0)\) defining the current state values on all state elements for cycle 0.

• A vector of primary input predicates \((\langle X^0, \ldots, X^k \rangle)\) defining the stimulus values for all primary input pins for each clock cycle of the error trace.

• A vector of primary output predicates \((\langle Y^0, \ldots, Y^k \rangle)\) defining the correct or expected values from the specification for at least one output pin across one or more clock cycles of the error trace.

The error trace can be written as follows:

\[
V^k_0 = (S^0, \langle X^0, \ldots, X^k \rangle, \langle Y^0, \ldots, Y^k \rangle) \tag{2.1}
\]

An error trace can be used to analyze failures and determine their root-cause. If the failure is due to a design error, the error trace can be used to determine which design components caused the mismatch. Likewise, if the failure is due to a verification error, the error trace can be used to analyze the verification environment. In this latter case, the verification environment usually caused a problem by generating the wrong stimulus or expected values in the error trace.

**Definition 4** A window of error trace \(V^k_p\) for clock-cycles \(p\) to \(q\) \((0 \leq p \leq q \leq k)\) is a subsequence of \(V^k_0\), defined as follows:

\[
V^q_p = (S^p, \langle X^p, \ldots, X^q \rangle, \langle Y^p, \ldots, Y^q \rangle) \tag{2.2}
\]

Where \(S^p\) defines the current state values on all state elements for cycle \(p\) when the sequential circuit is simulated with the error trace. It can be calculated by applying the initial state predicate and the first \(p\) primary input predicates to the sequential circuit transition relation, i.e., simulating the sequential circuit for \(p\) cycles.

Using this notation, a prefix window of length \(p\) for an error trace can be written as \(V^{p-1}_0\) and a suffix window of length \(k - p + 1\) can be written as \(V^k_p\). We will occasionally omit the term window and use the term suffix or prefix in place of suffix window or prefix window, respectively.
For this thesis, it is assumed that the error is first observed in the last clock cycle of an error trace. If this is not the case, a shorter error trace can be trivially generated by taking the shortest prefix window that exhibits an erroneous mismatch.

2.2.1 Iterative Logic Array

In this subsection, we introduce the concept of an Iterative Logic Array as well as some notational conventions.

In verification and debugging, it is necessary to model the behavior of a sequential circuit. This can be done using a method called the Iterative Logic Array (ILA), also known as time-frame expansion. Time-frame expansion models a sequential circuit by extracting the combinational part of a circuit and replicating it for the number of clock cycles that need to be modeled. Each copy of the combinational circuit is known as a time-frame.

To extract the combinational component, the current-state variables are treated as pseudo-inputs, the next-state variables are treated as pseudo-outputs and the state elements are removed. Next, the combinational component is replicated such that the next-state variables of time-frame \( i \) are connected to the current-state variables of time-frame \( i + 1 \).

Throughout this thesis, we will use a set of common conventions to enhance the readability of the text. For sequential circuit \( C \), let \( l \) denote the set of all nets in the circuit. Let \( x, y \) and \( s \) be a subset of these nets referring to the set of primary inputs, primary outputs, state elements respectively. For each \( z \in \{ x, y, s, l \} \), let \( z^i \) denote the corresponding Boolean vector for the \( i^{th} \) clock-cycle, or time-frame, for the operation of \( C \). Furthermore, let \( z^i_j \) denote the \( j^{th} \) indexed bit in the \( i^{th} \) time-frame for the Boolean vector \( z^i \). The transition relation for sequential circuit \( C \) is denoted by \( T(s^i, s^{i+1}, x^i, y^i) \), which is true if and only if given the current-state \( s^i \), applying primary inputs \( x^i \) to \( C \) will generate primary outputs \( y^i \) and the next-state \( s^{i+1} \). The replicated transition relation for the \( i^{th} \) clock-cycle will be denoted as \( T^i(s^i, s^{i+1}, x^i, y^i) \). For brevity, we will occasionally omit the parameters to the predicate when the context is clear.

Example 1 Figure 2.1 shows how to generate a time-frame expanded circuit. Figure 2.1(a) shows the original sequential circuit. First, the combinational component is separated from the
Chapter 2. Background

Chapter 2. Background

Sequential components by extracting the flip-flops, this is shown in Figure 2.1(b). The pseudo-inputs and pseudo-outputs are implicitly shown to be wires coming out of the dotted box. Next, the combinational component is replicated into a two time-frame expanded circuit shown in Figure 2.1(c).

Example 2 For the time-frame expanded circuit in Figure 2.1(c), the following is an error trace, annotated on the figure, demonstrating an erroneous behavior in the circuit:

\[ V_0 = \langle s_0^0, (x_0^0 \land x_1^1, x_0^1 \land x_1^1), \langle y_0^0, y_1^0 \rangle \rangle \]
Notice that in the second time-frame, the erroneous circuit generates $\overline{y_0}$ which is incorrect according to $V_{0}^0$. In addition, the prefix window $V_{0}^0 = \langle s_0^0, \langle x_0^0 \land x_1^0 \rangle, \langle y_0^0 \rangle \rangle$ and suffix window $V_{1}^1 = \langle s_1^0, \langle x_0^1 \land x_1^1 \rangle, \langle y_1^0 \rangle \rangle$ can be generated from this error trace.

### 2.2.2 Formal Verification

*Formal verification* is the process in which a design implementation is proved to be correct with respect to a formal property or specification, using formal methods \[48\]. Formal methods describe a broad set of mathematical techniques that, with respect to various logics and proof systems, allows one to prove properties about a system. These techniques are a type of *static verification* because they do not require a user to generate explicit input stimulus for the design. In contrast, *dynamic verification* generally refers to simulation-based techniques where explicit stimulus must be generated (e.g., directed tests or constrained random-tests). As such, formal verification has been used to great effect to complement dynamic verification techniques in order to improve overall design quality.

A popular type of formal verification is *model checking* (also known as formal property checking or simply property checking) \[35\]. In this scenario, an assertion in the form of a formal property is written based upon a high-level specification. Then, the RTL description is verified using a model checker to ensure that it does not violate any of the behaviors specified by the property. If the RTL description does not conform to the property, a counter-example is generated. In this scenario, a counter-example returned by the formal tool defines the initial state predicate as well as the primary input predicates. However, the primary output predicates are implicitly defined by the assertion by viewing the synthesized assertion as additional circuitry. Using this view, the final output signal of the synthesized assertion can be considered as a primary output pin which is expected to always output true.

### 2.3 Functional Debugging

*Functional debugging* (or logic debugging) is the process in which a failure is analyzed to dete-
mine its source(s). The source of the error can be in the design implementation or verification environment. In the former case, the term commonly used in literature \cite{128, 137} is design debugging, which has shown to be NP-Complete \cite{94}. In this thesis, we will use the term functional debugging interchangeably with the more generic term debugging for brevity. Additionally, the older term diagnosis is also equivalent to debugging, and may be used in places when referring to past literature.

With respect to design debugging, the error source(s) of a failure can be any component (e.g. gates, RTL source code, design modules) within the design implementation. The generic term suspect is used to specify such a component(s) (or equivalently location(s)) that can be the cause of the error as seen in the next few definitions.

**Definition 5** Let $E = \{F_1, F_2, \ldots, F_m\}$ be a set of $m$ functions corresponding to $m$ components in a sequential circuit with $n_1, n_2, \ldots, n_m$ output bits, respectively. And let $V$ represent an error trace exposing a failure in the corresponding sequential circuit. Then, $E = \{F_1, F_2, \ldots, F_m\}$ is a solution with respect to $V$, if and only if, there exists a set of $m$ functions, $\{G_1, G_2, \ldots, G_m\}$ with $n_1, n_2, \ldots, n_m$ output bits, that can replace $E$ such that $V$ no longer exposes a failure.

In other words, a solution is a set of components in a circuit that can be corrected in order to fix an observed failure for a single error trace.

**Definition 6** A component (or location) is said to be a suspect with respect to $V$, if and only if, it is part of a solution.

**Definition 7** The error cardinality (or cardinality) of a solution $E$ is defined to be the cardinality of the set i.e., $|E|$.

Notice that using these definitions, multiple solutions (e.g., $E_1, E_2, \ldots, E_k$) may exist for a single error trace. This is due to the fact that for a given error trace, there may be multiple equivalent ways to change the design to correct the failure \cite{137}. Some of these solutions will be different ways of synthesizing the actual fix\textsuperscript{2}, while others may be invalid when taking into

\textsuperscript{2}“actual” in the sense that this is a valid resolution to the underlying error which, in most cases, will be a fix which is consistent with the behavioral specifications.
account other error traces. This leads to the fact that multiple error traces can be used to refine the results by intersecting the corresponding solutions (for a given error cardinality) [128].

In most cases, we are only interested in minimum error cardinality solutions since non-minimum ones are trivially generated. For example, if \( \{F_1\} \) is a solution, then any set of components including \( F_1 \) are also solutions (e.g., \( \{F_1, F_2\} \), \( \{F_1, F_3\} \), \( \{F_1, F_4, F_5\} \) etc.). For this reason, unless otherwise specified, solutions will refer to minimum error cardinality solutions.

In certain situations, debugging algorithms may not return all possible solutions but instead a subset or superset of solutions. The following define terms for these situations.

**Definition 8** A debugging algorithm is said to be an under-approximation if and only if it returns a subset of all solutions for a given debugging problem.

**Definition 9** A debugging algorithm is said to be an over-approximation if and only if it returns a superset of all solutions for a given debugging problem.

Further, the relative number of solutions returned by a debugger is generally referred to as the resolution of the debugger. If a debugger returns relatively few solutions, then it is said to have good resolution. Otherwise, it has poor resolution. This qualitative measure depends not only on the specific debugging algorithm (e.g., use of over-approximations), but also the specific debugging instance (i.e., design and error trace).

### 2.4 Boolean Satisfiability

This section describes relevant background material on Boolean Satisfiability (SAT) that is necessary for understanding the concepts in this thesis. The following is a formal definition of the Boolean Satisfiability problem.

**Definition 10** Given a propositional logic formula \( \phi \) with \( n \) Boolean variables \( v_1, v_2, ..., v_n \), the Boolean Satisfiability problem asks whether there exists an assignment to its variables that evaluates \( \phi \) to true or whether no such assignment exists. If such an assignment exists then the problem is said to be satisfiable (SAT) otherwise unsatisfiable (UNSAT).
The Boolean Satisfiability problem has great historical importance as a variant of it was the first problem to be classified as NP-Complete \cite{40}. It also has great practical importance as many problems can be modeled as a SAT instance. Recent advances in modern SAT solvers have allowed them to solve industrial instances with millions of variables and clauses efficiently, making SAT solvers a popular tool for solving many VLSI problems \cite{17,39,61,91,103,104,118,123,126}.

### 2.4.1 CNF Representation

Modern SAT solvers accept SAT instances in conjunctive normal form (CNF), which is a conjunction of clauses, where each clause is made up of a disjunction of literals. The following example shows a simple SAT problem in CNF.

**Example 3** The following is a Boolean formula written in CNF.

\[
\phi = (a + \overline{b})(\overline{a} + b + \overline{c})(b + \overline{c})
\]  

(2.3)

\(\phi\) is SAT because \(\{a = 1, b = 0, c = 0\}\) is a satisfying assignment evaluating \(\phi\) to 1.

Logic circuits can be efficiently converted into CNF in linear time \cite{136} using the Tseitin transform by introducing auxiliary variables for each net in the circuit. Table 2.1 shows how common logic gates can be converted to CNF.

Besides simple logic gates, many higher level constraints in CNF can be generated using circuit constructs. One commonly used constraint is a cardinality constraint which ensures that for a set of Boolean variables, exactly \(N\) variables are set to 1. One space-efficient way to implement this is shown in Figure 2.2 with a tree of full adders that feed into a comparator. The input variables \((s_1, \ldots, s_n)\) are added together, and compared to a constant \(N\). Since the output of the comparison operator is constrained to exactly 1, this forces the input variables to have exactly \(N\) 1s. This encoding is linear in the number of variables input into the cardinality constraint \cite{128}. The tree of adders and comparator can be formulated in CNF by using simple logic gates from Table 2.1.
### Chapter 2. Background

#### 2.4.2 Unsatisfiable Cores and Proofs of Unsatisfiability

The following discusses some concepts revolving around UNSAT Boolean formulas that relate to the material in this thesis.

**Definition 11** Given an unsatisfiable (UNSAT) Boolean formula $\phi$ in conjunctive normal form (CNF), an UNSAT core is a subset of clauses in $\phi$ that are unsatisfiable.

**Definition 12** Given an unsatisfiable (UNSAT) Boolean formula $\phi$ in conjunctive normal form (CNF), a Minimal Unsatisfiable Subset (MUS) is an UNSAT core where every proper subset of clauses is satisfiable (SAT).

<table>
<thead>
<tr>
<th>Gates</th>
<th>CNF</th>
</tr>
</thead>
</table>
| $y = AND(x_1, x_2, ..., x_n)$ | $(x_1 + \overline{y})(x_2 + \overline{y})...(x_n + \overline{y})$  
|         | $(\overline{x_1} + x_2 + ... + x_n + y)$                              |
| $y = NAND(x_1, x_2, ..., x_n)$ | $(x_1 + y)(x_2 + y)...(x_n + y)$                                         
|         | $(\overline{x_1} + \overline{x_2} + ... + \overline{x_n} + y)$            |
| $y = OR(x_1, x_2, ..., x_n)$    | $(\overline{x_1} + y)(\overline{x_2} + y)...(\overline{x_n} + y)$        
|         | $(x_1 + x_2 + ... + x_n + y)$                                           |
| $y = XOR(x_1, x_2)$            | $(\overline{x_1} + x_2 + y)(x_1 + \overline{x_2} + y)$                  
|         | $(x_1 + x_2 + \overline{y})(\overline{x_1} + \overline{x_2} + \overline{y})$ |
| $y = XNOR(x_1, x_2)$           | $(\overline{x_1} + x_2 + \overline{y})(x_1 + \overline{x_2} + \overline{y})$  
|         | $(x_1 + x_2 + y)(\overline{x_1} + \overline{x_2} + y)$                  |
| $y = NOT(i)$                    | $(i + y)(\overline{i} + \overline{y})$                                    |
| $y = BUFFER(i)$                 | $(i + \overline{y})(\overline{i} + y)$                                   |
| $y = MUX(s, x_1, x_2)$          | $(x_1 + \overline{y} + s)(\overline{x_1} + y + s)$                      
|         | $(x_2 + \overline{y} + \overline{s})(\overline{x_2} + y + \overline{s})$ |

Table 2.1: CNF for Basic Logic Gates
In a given instance, there may be multiple UNSAT cores and MUSs since there can be many different subsets of the problem that are UNSAT. The following example illustrates this concept.

**Example 4** The following is an UNSAT Boolean formula written in CNF since there is no satisfying assignment to the variables that can make the formula true.

\[
\phi = (a + \overline{b})(\overline{a} + \overline{b} + \overline{c})(a + \overline{c} + d)(\overline{a} + d)(b + \overline{c})(c)(\overline{c} + \overline{d}) \tag{2.4}
\]

The following is an UNSAT core which is also a MUS:

\[
U_1 = (a + \overline{b})(\overline{a} + \overline{b} + \overline{c})(b + \overline{c})(c) \tag{2.5}
\]

Another UNSAT core which is also a MUS, is given by the following:

\[
U_2 = (a + \overline{c} + d)(\overline{a} + d)(\overline{c} + \overline{d}) \tag{2.6}
\]

With respect to debugging, a MUS (and to some extent an UNSAT core) intuitively represents one way in which an error trace can excite an error, traverse its effects through the design components and cause a failure at the observation points. In this view, clauses correspond to the initial state, input stimulus, components of the design and expected output values.
The resolution rule is an inference rule that produces a new clause that is implied by two source clauses with literals of opposite polarity. In detail, if given two clauses \((a_1 + \cdots + a_n + c)\) and \((\bar{c}+b_1+\cdots+b_n)\), we can imply the clause \((a_1 + \cdots + a_n + b_1 + \cdots + b_n)\). This operation can also be described as resolving on literal \(c\). This rule is the basis for one the earliest algorithms [44] developed to solve SAT instances. The central idea in that algorithm is that one resolves on a variable \(c\) for all combinations of original clauses. This will imply many new clauses that contain the information from the original clauses involving \(c\). If one removes the original clauses involving \(c\), and adds the newly implied ones, then the resulting instance is SAT if and only if the original instance is SAT.

**Example 5** In the following equation \(a\) and its complement exist in the two original clauses. A third clause can be implied using the resolution rule, which takes a disjunction of all the other literals in the original clauses to produce the new one.

\[
(a + b)(\bar{a} + c) \rightarrow (b + c)
\]

A resolution graph [19] is a tree demonstrating how clauses in an UNSAT core can be combined using the resolution rule to generate the empty clause. This serves as a proof of unsatisfiability (also known as a resolution refutation) of a SAT instance. The root nodes of the graph are the original clauses, the intermediate nodes correspond to implied or learnt clauses of the SAT instance and the leaf node is the empty clause. In the worst case, a resolution graph can be exponential in the size of the original SAT instance [136].

Modern SAT solvers can generate a proof of unsatisfiability along with a corresponding resolution graph that shows that a SAT instance is unsatisfiable [146]. This requires keeping track of all resolution steps performed to generate the learnt clauses, which can add significant overhead to the solving process. However, if this information is not retained, a partial proof can be obtained that shows how a combination of original and learnt clauses can be resolved to generate the empty clause. We denote this partial proof graph as the conflict graph. These graphs can be used for further analysis and, with additional computation, can be used to generate an approximation. This is used to great effect in many verification and debug algorithms [22, 30, 37, 100].
Example 6 The following is an UNSAT Boolean formula written in CNF.

\[ \phi = (d)(a + b)(\overline{a} + \overline{b})(a + \overline{c})(\overline{a} + c)(b + \overline{d})(c + \overline{d})(\overline{b} + \overline{c} + d) \]

An UNSAT core, \( U \), of \( \phi \) is: \( (d)(\overline{a} + \overline{b})(a + \overline{c})(b + \overline{d})(c + \overline{d}) \). By resolving these clauses together, a resolution graph can be generated as shown in Figure 2.3, where the empty clause is denoted by “( )”. Each root node in the graph represents an original clause in \( U \), while each intermediate node represents a clause that can be implied, or learned from the original clauses. The UNSAT core implies the empty clause resulting in an UNSAT instance.

A conflict graph can be any subset of the graph in Figure 2.3 that contains the empty clause.

2.5 SAT Solvers

Most modern SAT solvers \[18, 49, 53, 107\] use a variation of the search-based Davis Putnam Logemann Loveland (DPLL) algorithm \[43\] called Conflict-Driven Clause Learning (CDCL). The basic form of the search-based DPLL algorithm is shown in Algorithm 2.1. The SAT problem can be thought of as a binary tree where each level represents a variable, each internal node represents a partial assignment and each edge represents an assignment to true or false.
Algorithm 2.1 The DPLL algorithm

1: procedure DPLL_SAT(Partial Assignment A)
2: \hspace{1em} if contains_unsatisfied_clauses() then
3: \hspace{2em} return UNSAT
4: \hspace{1em} else if all_clauses_satisfied() then
5: \hspace{2em} return SAT
6: \hspace{1em} else
7: \hspace{2em} v ← next_unassigned_variable()
8: \hspace{2em} return DPLL_SAT(A ∪ (v=0)) ∨ DPLL_SAT(A ∪ (v=1))
9: \hspace{1em} end if
10: end procedure

The DPLL algorithm can be thought of as doing a depth first search along this tree.

At each step of the algorithm, it performs two checks. One to see if there are any unsatisfied clauses and returns UNSAT immediately. The other checks to see if all the clauses are satisfied and returns SAT. Otherwise, it will pick a variable that has not been assigned yet and recursively call itself to explore another node in the tree. The algorithm will eventually result in every possible branch being explored, making this algorithm complete. However, due to this fact, the worst case run-time behavior of this algorithm will be exponential in the number variables.

Although CDCL algorithms do not improve upon this worst case behavior, they add significant benefit over the basic DPLL algorithm and show a large improvement on a wide variety of problems. There are three major areas that CDCL solvers have improved upon over DPLL algorithms: decision heuristics, Boolean constraint propagation (BCP) and conflict analysis.

The first improvement is the order in which variables are decided upon, also known as decision heuristics. Since the underlying algorithm is search-based, the order in which variables are selected is extremely important. Most modern solvers use a variation of the variable state independent decaying sum (VSIDS) decision heuristic which was used in zChaff [107]. This heuristic gives preference to literals that appear most often in the SAT problem while giving heavier weight to recently discovered information. One important aspect of this heuristic is that it keeps running totals of the literal counts which can be computed very efficiently.
The second improvement, BCP, relates to the process of determining assignments to variables that are implied by the current partial assignment. A \textit{unit clause} is produced if, in the current partial assignment, all but one literal in a clause evaluates to zero and the remaining literal is unassigned. For this type of clause, the remaining literal must be set to \textbf{true} or else the clause will not be satisfied. However, setting the literal to \textbf{true} may cause other clauses to become unit clauses and the process can repeat. This process can cause a single decision to extend the partial assignment by a large number of literals, significantly speeding up the search (and conflict generation) process. This allows huge amounts of the search tree to be pruned. In fact, the authors of \cite{107} estimate that 90\% of the run-time is spent inside this one routine. Other anecdotal evidence suggests a higher percentage. Major improvements to Boolean constraint propagation involve engineering efficient implementations by choosing the right data structures and associated algorithms.

\textbf{Example 7} Consider the SAT instance with the partial assignment $a = 0$:

$$(a + b)(\overline{b} + \overline{c})(\overline{c} + \overline{d})$$

Since $a = 0$, the clause $(a + b)$ becomes a unit clause and we can imply that $b = 1$. But this makes $(\overline{b} + \overline{c})$ a unit clause, implying that $c = 0$. This in turn, implies that $d = 0$. All clauses are now satisfied, so the instance is SAT by only deciding on one variable and running Boolean constraint propagation.

The third major improvement relates to the way in which modern algorithms handle a conflict found in Boolean constraint propagation. When a conflict arises, in the basic DPLL algorithm, the search backtracks one level up the tree and tries the other branch. A major contribution in this area involves a smarter way to “learn” and backtrack based on the decisions and implications from BCP made up to this point in the algorithm. This occurs in the conflict analysis procedure.

During conflict analysis, an implication graph is built that shows how each variable in the current assignment was derived. The implication graph is a directed acyclic graph where vertices represent an assignment to a variable and its decision level, and the edges are annotated with
the clause used to derive the fanout node. The possible sources for the assignment on each vertex are either a variable that was assigned through a decision, or through an implication by BCP(). A conflict occurs when a variable is implied to be both true and false at the same time. This implies that the current partial assignment can never yield a SAT result and must be reversed by backtracking. To ensure that the search does not enter this unsatisfiable subspace at a later point, a clause is added to (or learned by) the solver. This learnt clause has the property that it is implied by the original SAT instance. It can be derived by taking a cut of the implication graph separating the conflict and any connected decisions. The clause literals are found by taking the negation of the literals associated with the cut-edges. A solver may add clauses derived from any cut since they are implied by the original instance. However, most modern solvers use a heuristic that only adds a single clause corresponding to the first cut involving the most recent decision variable [19].

**Example 8** Consider the following SAT instance:

\[ \phi = C_1 \land C_2 \land C_3 \land C_4 \land C_5 \land C_6 \land \ldots \]

\[= (x_1 + x_2 + \bar{x}_3)(x_1 + \bar{x}_4)(x_3 + x_4 + x_5)(\bar{x}_5 + x_6)(\bar{x}_5 + x_7 + \bar{x}_8)(x_6 + x_8)\ldots\]

and the partial assignment of \(x_7 = 0\) at decision level 2, \(x_2 = 0\) at decision level 3, and \(x_1 = 0\) at decision level 5. Figure 2.4 shows resulting implication graph with several different cuts. Each cut separating the conflict and its derivative decisions will result in a new learnt clause preventing future searches into this subspace. The learnt clauses from the respective cuts in the figure are listed here:

- **Cut1** : \((\bar{x}_5 + x_8)\)
- **Cut2** : \((\bar{x}_5 + x_7)\)
- **Cut3** : \((x_3 + x_4 + x_7)\)
- **Cut4** : \((x_1 + x_2 + x_7)\)

Each learnt clause can be derived from taking the negation of the conjunction of the associated literals from the cut.
After a clause is learned, backtracking occurs which removes all variable assignments up to the most recent decision level involved in the conflict. Due to the learnt clauses, this enables non-chronological backtracking, allowing the search process to backtrack to decision levels beyond the most recent decision. Both conflict analysis and non-chronological backtracking have shown to be very effective in pruning out large parts of the search space resulting in a major improvement to modern SAT solvers.

Algorithm 2.2 shows a modern variant of CDCL algorithm that most SAT solvers use. The sub-routines used by this algorithm corresponding to the major improvements mentioned above: decide() corresponds to decision heuristics, bcp() corresponds to Boolean constraint propagation, analyze_conflict() corresponds to conflict analysis, and backtrack() corresponds to non-chronological backtracking.

From Algorithm 2.2, the algorithm first attempts to propagate any unit clauses under the current assignment using the bcp() procedure on line 3. If this does not yield a conflict then either all variables are assigned in which case the problem is SAT, or a new variable has to be decided upon on line 8. If BCP does yield a conflict then the analyze_conflict() procedure is run to generate a conflict clause. At this point, if no decisions have been made, the conflict is implied by the original problem and the overall procedure returns UNSAT. Otherwise, the backtrack() procedure on line 15 reverses the most recent decision involved in the conflict.
Algorithm 2.2 Modern SAT algorithm

1: procedure Modern_SAT

2: while true do

3: bcp() // Propagate unit clauses

4: if no conflict then

5: if all variables assigned then

6: return SAT

7: else

8: decide()

9: end if

10: else

11: analyze_conflict() // Analyze conflict and add conflict clause

12: if top-level conflict found then

13: return UNSAT

14: else

15: backtrack() // Undo decision assignments until conflict clause is unit

16: end if

17: end if

18: end while

19: end procedure

A variation of a SAT solver is an all-solution SAT solver which will find all satisfying assignments to a SAT instance. A regular SAT solver can be easily extended by adding a blocking clause which is a disjunction of the negation of all the assigned literals. This prevents the same satisfying assignment from being found again. After this satisfying assignment is added, the solver is run again to find the next one. When the solver returns UNSAT, all possible satisfying assignments have been found.

An important practical point to consider is that in many cases, a full assignment to all of the variables in a SAT instance is not necessary. Instead, only certain variables are regarded
as important. In these cases, the blocking clause should contain only the literals from the important variables. Many modern solvers have this feature built-in to allow use with practical problems.

### 2.5.1 Incremental SAT and Assumptions

Modern SAT solvers support a feature called incremental SAT \[50\]. Incremental SAT has been used to great effect in many different verification applications \[51, 52, 125\]. In general, this term refers to two features:

- **Adding Additional Clauses**: This feature allows one to add additional clauses to the SAT solver (thus changing the instance) after one or more calls have been made.

- **Solving with assumptions**: This feature allows one to solve a given SAT instance with assumptions in the form of additional unit clauses. Any SAT assignment returned when using assumptions will guarantee that these unit clauses are respected. An UNSAT result means that no such assignment with these unit clauses is possible with the given instance.

The first feature of adding additional clauses allows efficient solving of problems where one may want to incrementally build and solve a SAT instance. A particular application of this feature is finding all SAT solutions. After the first solve call, a blocking clause may be added and another solve call can be made. The benefit of this implementation is that learnt clauses can be re-used so that work does not need to be repeated.

The second feature of assumptions is beneficial when one wants to solve variations of a base SAT instance. An example of this may be simulating a combinational circuit under different input conditions. An important guarantee when using assumptions is that any clause that is learnt will only be implied from the original clauses and never the assumptions. Thus, it is safe to call the solver again with a different set of assumptions.

### 2.6 SAT-based Automated Design Debugging

This section briefly describes background and notation for SAT-based design debugging that is relevant to the contributions in this thesis. SAT-based design debugging \[128\] is an algorithm
that encodes the design debugging problem into a SAT instance for a given error trace and number of errors. This construction formally models the time-framed expanded design, error trace and additionally, error models to identify locations of potential functional errors. Due to the output value(s) mismatch between the design and expected outputs for a given error trace, this instance will be UNSAT when all error models are off. However when an error model is on, it effectively replaces the associated component with an arbitrary non-deterministic function. If this non-deterministic function can fix the observed failure, then the entire instance will be SAT. Thus, each satisfying assignment of this SAT instance corresponds to a debugging solution i.e., a set of suspects that can potentially explain the observed failure.

The SAT instance for sequential circuit $C$ and error trace $y^h_0$ is constructed in several steps. First, an error model is added for each design component (e.g., gate, module, etc.) by augmenting the corresponding clauses in its CNF encoded transition relation $T$. This is done by generating a new suspect variable ($e_i$), and adding it to each clause corresponding to that design component. When a suspect variable is true, it satisfies all the clauses of the associated component allowing its outputs to be free, effectively encoding an arbitrary non-deterministic function. Otherwise, the component’s behavior remains unchanged. We denote this enhanced transition relationship by $T_{en}$. Note that this technique can be used to model RTL errors by adding one suspect variable to each clause corresponding to a RTL construct such as a Verilog assignment or module [11]. The mapping between RTL construct and clauses can be obtained by using a standard RTL frontend to translate RTL to gates, and subsequently gates to clauses.

Next, $T_{en}$ is replicated for the length of the error trace (i.e., time-frame expanded), where each copy is labeled $T_{en}^i$ for the $i^{th}$ copy. During this process, the suspect variables are not replicated because each one corresponds to a single, potentially erroneous, design component regardless of the underlying time-frame. Additionally, constraints for the initial state ($S^0$), vector of inputs ($\langle X^0, \ldots, X^k \rangle$) and vector of expected or correct outputs ($\langle Y^0, \ldots, Y^k \rangle$) from the error trace are added.

---

3Due to the use of non-deterministic functions, a satisfying assignment may not translate to a deterministic function at the associated component(s). This theoretically means that this function is not synthesizable because it is non-deterministic. However, in practice, this is a non-issue since this situation is exceedingly rare and results in a negligible increase in the number of solutions returned.
Finally, cardinality constraints \([\Phi(N)]\) (denoted by \(\Phi(N)\)) are used to limit the number of active suspect variables to exactly \(N\), indicating the search for exactly \(N\) simultaneous errors in the circuit.

Given an error trace \(V^k_0\) or a window of an error trace \(V^q_p\), design debugging can be encoded by the following SAT problems, respectively:

\[
egin{align*}
\text{Debug}^k_0(N) &= S^0 \land \phi_N(E) \land \left( \bigwedge_{i=0}^{k} X_i \land Y_i \land T^i_{en} \right) \\
\text{Debug}^q_p(N) &= S^p \land \phi_N(E) \land \left( \bigwedge_{i=p}^{q} X_i \land Y_i \land T^i_{en} \right)
\end{align*}
\]

In the above equation, we will occasionally omit the parameter for error cardinality \(N\) for brevity.

Note that for \(N = 0\), \(\text{Debug}^k_0\) is UNSAT, since the error trace applied to the design without any active error suspect variables cannot produce the expected outputs defined in the error trace. If this is not the case, the equation is trivially satisfiable when \(N = 0\) (and thus for all \(N >= 0\)) indicating that the failure cannot be observed in the modeled window and an extended error trace must be used.

In a satisfying assignment of Equation 2.7, each active suspect variable corresponds to a suspect, and the set of active suspect variables correspond to a debugging solution. To find all such solutions, for each satisfying assignment, a blocking clause is added to the debugging instance to block the active suspect variables from appearing again as a satisfying assignment. Using incremental SAT, this updated instance can be sent back to the solver to find another solution. When the solver eventually returns UNSAT, all possible solutions have been found.

**Example 9** Figure 2.5 shows a two time-frame expanded circuit of an erroneous two gate design with one state element. The symbol \(\otimes\) denotes an error model where the associated suspect variable \((e_1, e_2)\) is labeled beside it. The incorrect gate is \(g_2\) which should be a buffer instead of an inverter. The error trace:

\[
V^1_0 = \langle s^0_0, (x_1^0 \land x_2^0, x_1^1 \land x_2^1), (y_1^1 \land y_2^1) \rangle
\]

demonstrates an erroneous behavior of the circuit. A SAT instance in CNF for this debugging
Figure 2.5: SAT-based Design Debugging

problem can be written as such:

\[(s^0_0)(x^0_1)(x^0_2)\]
\[(s^0_0 + y^0_1 + e_1)(x^0_1 + y^0_1 + e_1)(s^1_0 + x^1_1 + y^1_1 + e_1)\]
\[(x^0_2 + y^0_2 + e_2)(\overline{x^0_2} + \overline{y^0_2} + e_2)\]
\[(x^1_1)(x^1_2)\]
\[(s^1_0 + y^1_2)(s^1_0 + y^0_2)\]
\[(s^0_0 + \overline{y^1_1} + e_1)(x^1_1 + \overline{y^1_1} + e_1)(s^1_0 + \overline{x^1_1} + y^1_1 + e_1)\]
\[(x^1_2 + y^1_2 + e_2)(\overline{x^1_2} + \overline{y^1_2} + e_2)\]
\[(y^1_1)(y^1_2)\]
\[(e_1 + e_2)(\overline{e_1} + \overline{e_2})\]  \hspace{1cm} (2.8)

For \(N = 1\), a satisfying assignment for the suspect variables \(\{e_1, e_2\}\) is \(\overline{e_1} \land e_2\). Adding the blocking clause \((\overline{e_2})\) to the problem causes it to be \(\text{UNSAT}\). This implies that \(\{g_2\}\) is a solution, and \(g_2\) is the only gate that can be modified to correct the erroneous behavior.
2.7  Formal Properties and SystemVerilog Assertions

A formal property is a precisely defined mathematical statement representing the behavior of a formally defined system. In the context of verification, these properties are usually described in some temporal logic \[15, 115\] in order to accurately represent complex behaviors of a system over time. Formal properties have been used to great effect in many applications such as system synthesis \[20\], coverage \[114\], constraints for random simulation \[145\], and model checking \[35\]. The two dominant languages used to specify these properties are SystemVerilog assertions (SVA) \[1\] and Property Specification Language (PSL) \[3\].

During verification, formal properties are typically used either as assertions or assumptions. Assertions specify a property about a system that should always be true. The typical use cases for assertions are to either run them during simulation to check that the properties have not been violated, or to use a model checker and prove that a system conforms to the property. In contrast, assumptions specify a constraint on the system, typically on the inputs, to restrict the state space of the design. These are typically used in conjunction with a formal tool to model the environment outside of a design. For this thesis, formal properties will only be discussed with respect to assertions and assumptions, thus the term property may be used interchangeably with these terms when the context is clear.

This rest of this section gives a brief overview of the SystemVerilog assertion language and related concepts due to its relevance with the contributions in this thesis. For a more detailed treatment please refer to \[1, 140\].

SVA is a concise language for describing temporal system behavior for use in verifying RTL designs. Each system behavior can be specified by writing a SVA property which in turn can be used as either an assertion or assumption.

Properties are derived from several different types of expressions that build upon each other. The simplest type of expression is a Boolean expression. Since each assertion or assumption has a clock event, these expressions are sampled just before the clock to ensure consistent behavior across tools. A sequence expression specifies the behavior of Boolean expressions over one or more clock cycles using operators such as delays and repetitions. These can be combined to
define complex sequence expressions that describe multiple linear sequences known as *threads*. For example, `start #[1:3] stop` specifies three separate threads of `start` followed by `stop` with varying lengths of delay in between. If one of the threads evaluates to `true` then the sequence is said to *match* i.e. evaluates to `true`. Threads allow great ease in specifying common design behaviors but also lead to increased difficulty in debugging. A *Property expression* specifies more complex combinations of sequence expressions such as implications. A grammar of common SVA operators is listed in Table 2.2.

The implication property operator (`|-|`, `|=|`) is similar to an if-then structure. The left-hand side is called the *antecedent* and the right-hand side is called the *consequent*. With respect to the implication operator, a property is said to be *vacuously true* if every evaluation of the antecedent is false. Example 11 gives an overview of how several common SVA operators are used.

**Example 10** The following show a sample of SystemVerilog assertions and their respective specifications in comments:

```// Boolean expressions:
```

---

---

The noun that defines this situation is known as *vacuity*.
// Requires 'data' to be equivalent to 'correct_data'
data == correct_data

// Ensures 'Read' not high or 'Write' not high
!(Read && Write)

// Requires 'a' in this cycle be equal to the value of 'b' in the last cycle
a == $past(b)

// Requires 'state' be one-hot encoded
$onehot(state)

// Sequential expressions:
// 'a' is high and one cycle later 'b' is high
a ##1 b

// 'a' is high and between one and three cycles later 'b' is high
a ##[1:3] b

// 'a' is repeated for three or four clock cycles
a[*3:4]

// 'a' followed by one or more 'b's followed by 'c'
a ##1 b[1:*] ##1 c

// 'a' followed by two, not necessarily consecutive, occurrences
// of 'b', followed by 'c'
a ##1 b[->2] ##1 c

// Equiv. to (a ##2 b ##1 a ##2 b)
(a ##2 b)[*2]

// Property Expressions:
// If 'a' then 'b' must occur in the same cycle
a |-> b

// If 'a' then 'b' must occur one cycle later
a |=> b
// 'q' increments, provided that 'en' is high

\[ en \implies q = \text{past}(q+1) \]

**Example 11** Consider the specification: “If `start` is active, `data` repeats 2 times before `stop` and one cycle later `stop` should be low.” A waveform that conforms with this specification is given in Figure 2.6 with respect to a clock named `clk`. This can be concisely written as the SVA property:

\[ \text{start} \implies \text{data}[^2] \#1 \text{stop} \#1 \neg \text{stop} \]

### 2.8 Summary

This chapter presents background material necessary for understanding the contributions of this thesis. First, definitions and concepts regarding functional verification and debugging are presented. Next, a brief discussion on Boolean Satisfiability, SAT solvers, and SAT-based design debugging is given. Finally, formal properties and SystemVerilog assertions are described.
Chapter 3

Core-Guided Abstraction and Refinement

3.1 Introduction

Automated design debugging techniques have shown great promise in reducing the tedious, manual nature of design debugging. Over the past ten years, these techniques have made significant progress in their ability to handle large instances. Unfortunately, the design size – a key parameter of debugging complexity – has also seen unrelenting growth, limiting the adoption of automated debugging techniques in industrial environments. This chapter introduces a novel abstraction and refinement algorithm for automated design debugging that manages this key parameter of debugging complexity. This technique provides a sound and complete method for analyzing large design debugging instances in a SAT-based framework, providing significant benefit in run-time and memory over previous techniques.

Abstraction and refinement \[34, 35\] in the context of model checking has provided a scalable and robust method for tackling large industrial instances. It is generally considered one of the major advancements in the field with numerous novel extensions and derivative works \[30, 36, 99\]. This general methodology has two broad phases. The first phase generates an abstraction which represents a simplified approximation of the original concrete problem. Ideally, this abstraction is a strong enough approximation to perform the desired analysis. However if it
is not, the second phase occurs, which iteratively refines the abstract problem into a stronger approximation with respect to the original problem. After a sufficient number of refinement iterations, the model contains enough of the original problem such that a complete analysis can be performed.

Previous work in automated design debugging \cite{120, 121} has used a similar approach to manage the design size by abstracting nodes in the design to generate a simpler abstract debugging problem. However, one of the key limitations of this prior work is that to achieve complete results, the refinement technique requires an increased error cardinality due to the fact that a single error may be mapped to multiple ones in the abstract model. This approach guarantees that the algorithm will return a superset of the error locations that would have been found in the concrete model but at the cost of high error cardinality. This inevitably has a direct impact on the technique’s performance partially negating its benefit.

This chapter introduces a robust sequential abstraction and refinement algorithm for design debugging that manages the design size without the need to increase the error cardinality that impacts the underlying complexity exponentially \cite{137}. By managing this key parameter of the debugging complexity without the need for an increase in the error cardinality, the problem size can be significantly reduced to allow larger instances to be solved.

The algorithm begins by generating an abstract problem that is a strict under-approximation of a SAT-based debugging formulation. Using simulated values of the erroneous circuit, modules of the design are abstracted, reducing the complexity of the design size. After the abstracted problem is solved, the refinement strategy uses UNSAT cores to direct which modules are to be refined, avoiding the need for a costly increase in the error cardinality. An extension of this refinement strategy is also presented in which UNSAT cores are utilized to direct not only which modules should be refined, but also at what clock cycles in the error trace they should be refined. This provides the auxiliary benefit of managing the error trace length in addition to design size.

Experimental results show the efficacy of this technique compared to previous work on large industrial hardware designs with long error traces from OpenCores \cite{111} and from our industrial partners. The proposed abstraction and refinement technique is able to return solutions for 19
of the 22 instances compared to only 9 when the technique is not used. The extension provides further benefits as all 22 instances return solutions at the cost of increased refinement iterations.

The remaining sections of the chapter are organized as follows. Section 3.2 provides notation and background information. The proposed abstraction and refinement technique is presented in Section 3.3. Section 3.4 presents experimental results and Section 3.5 concludes this chapter.

### 3.2 Preliminaries

#### 3.2.1 Module Notation

As described in Section 2.2.1, a sequential circuit $C$ can be represented by a transition relationship, $T(s^i, s^{i+1}, x^i, y^i)$. This transition relation can be written in terms of modules or components corresponding to subcircuits of $C$ (e.g. a Verilog module) such that the outputs of a module are only a function of its inputs. Given module $q$ and time-frame $i$, let $mx^i_q, my^i_q \subseteq l^i$ (recall $l$ represents the set of all nets in the circuit) denote the corresponding Boolean vector of its module inputs and outputs, respectively. A module, $M_q(mx^i_q, my^i_q)$, describes constraints such that when applying inputs $mx^i_q$, outputs $my^i_q$ will be generated. Logic that is not contained within any module is denoted by $\phi_{\text{glue}}$. This definition can be applied recursively for each level of the design hierarchy. Using this notation, the transition relation can be expressed as:

$$T(s^i, s^{i+1}, x^i, y^i) = \phi_{\text{glue}}(l^i) \land \bigwedge_{q \in \text{modules}} M_q(mx^i_q, my^i_q)$$ (3.1)

The error trace can also be used to simulate $C$ and generate internal simulated values for all nets. In particular, they can be used to generate a predicate $MX^i_q$ and $MY^i_q$ corresponding to a module $q$’s input and output values for time-frame $i$.

The sequential circuit in the following example will be used throughout this chapter to illustrate the proposed abstraction and refinement scheme. It presents the SAT-based debugging formulation (from Equation 2.7) of a sequential circuit containing modules.

**Example 12** Figure 3.1(a) shows a simple sequential circuit with five gates and two state elements. The circuit has two modules, A and B. A three cycle error trace exposing an erroneous
behavior in this circuit is given by:

\[ V_0^2 = \langle s_0^0 \wedge s_1^1, \langle x_0^0 \wedge \overline{x_1}, x_0^1 \wedge x_1^1, \overline{x_0} \wedge \overline{x_1} \rangle, \langle y_0^0 \wedge y_0^1 \wedge y_0^2 \rangle \rangle \]

Figure 3.1(b) shows the SAT-based debugging formulation unrolled for three time-frames for the given circuit and error trace where \( \otimes \) denotes the error model. The suspect variables for each error model are denoted by \( e_i \) corresponding to gate \( g_i \). For \( N = 1 \), the gates \( g_0, g_2, g_3, g_4 \) will be returned as functionally equivalent suspects that could fix the observed failure.
3.2.2 Suspect-Guided Abstraction and Refinement

Abstraction and refinement techniques have been widely used in model checking to overcome the state-space explosion problem [34, 36, 74, 99]. Roughly speaking, these techniques abstract details of the original concrete problem and create a simpler abstract problem. The abstract problem is solved and depending on the results, either a refinement step is necessary or the result is complete and the algorithm terminates.

Abstraction and refinement has also been applied to design debugging, which aims to tackle the design size aspect of debugging complexity in [121]. We will denote this technique as suspect-guided abstraction and refinement due to its use of suspects for refinement.

The initial abstraction in this technique simplifies the design by removing all internal logic of components associated with the current abstraction level. In practice, this initial abstraction corresponds to removing all logic within each of the top level RTL modules (e.g., removing the constraints for each $M_q(x^i_q, y^i_q)$). For each of these components, its associated outputs are replaced by primary inputs, where its corresponding stimulus values are defined by the module output predicates as described in Section 3.2.1.

After these operations, a new abstract debugging problem is defined with the simplified sequential circuit along with the error trace, which is augmented with module output values. This abstract debugging problem can then be analyzed by any debugging algorithm to return solutions. A necessary point in this formulation is that a potential suspect is available for each of the corresponding abstracted modules. Once the debugger analysis is complete, the debugger will return a set of solutions. If any of the abstracted modules is contained in the returned solutions, then that module is refined. This entails adding back all the logic within that module. Optionally, the next hierarchical level within that module may be similarly abstracted in a recursive manner. This refinement step produces a new circuit (and associated error trace), defining the abstract debug problem for the next iteration. This process repeats until no more new modules are refined.

To achieve complete results, each call to the debugger must increase error cardinality in the abstract problem up to the total number of potential suspects. This adversely affects the performance of the debugger because the solution space grows exponentially with the error
cardinality \(137\). This step is necessary because a single error in the concrete problem may map to multiple errors in the abstract counterpart. This can occur if the propagation path of an error passes through multiple abstract nodes requiring a higher cardinality solution. The following example demonstrates this issue.

Example 13 Figure 3.2(a) shows a simple combinational circuit with five gates. A single buffered input fans out to three inverters, which then reconverges to a single AND gate. Error models are placed on the four internal modules labeled \(A, B, C, D\). For \(N = 1\) error cardinality, the only solution that exists is \(\{A\}\).

Figure 3.2(b) shows the initial abstraction using suspect-guided abstraction and refinement where modules \(A, B, C, D\) are abstracted. If only \(N = 1\) cardinality is analyzed, then there are no solutions. To ensure all suspects are found, error cardinalities up to \(N = 4\) must be analyzed. The only solution that can be found is at \(N = 3\) which is \(\{B, C, D\}\). Thus these modules are refined for the next iteration. In the next iteration, modules \(B, C, D\) are no longer abstracted, and there is one \(N = 1\) solution which is \(\{A\}\) (in addition to the same \(N = 3\) solution). Since \(A\) is the only abstracted module, it is refined. Finally, the problem is analyzed once again and \(\{A\}\) is found as the only \(N = 1\) solution.

3.3 Abstraction and Refinement

This section describes the proposed abstraction and refinement algorithm that addresses a key component of the debugging complexity, namely the design size, without the need to increase the error cardinality. The abstraction works by generating a strict under-approximation of the concrete debugging problem by applying predicates to the inputs and outputs of modules using the simulated values from the erroneous circuit. These values allow the constraints of the modules to be removed, thus simplifying the problem and effectively managing its size. After the abstract problem is solved for all solutions, the refinement strategy utilizes the resulting UNSAT core to determine which modules need to be refined. This analysis may return multiple modules in a given iteration effectively removing the need to increase the error cardinality. Finally, an extension of this refinement strategy is presented in which the UNSAT core is
utilized to decide both the modules and time-frames to be refined. This provides the additional benefit of refinement across time, effectively managing the error trace length.

### 3.3.1 Abstraction Formulation

The abstract instance is generated by starting with the concrete formula for SAT-based debugging from Equation 2.7 (Section 2.6) and adding selected constraints that will simplify it. The simplified formula offers the benefit of reducing the solve time and memory footprint compared to the concrete one. The key insight in this formulation is that the initial state and input vectors in the error trace can imply a great deal of additional information. The initial state and input stimulus, along with the erroneous circuit, provide the ability to generate the simulated value of any internal net in the erroneous circuit at any time-frame. We will use these values to constrain the debugging formula to generate an abstract instance.

We create an abstract debugging instance by applying, for each module $q$, the simulated
input \( (MX^i_q) \) and output \( (MY^i_q) \) predicates to Equation 2.7:

\[
InitAbsDebug^k_0 = S^0 \land \Phi_N \land \bigwedge_{i=0}^{k} X^i \land Y^i \land T^i_{en} \land \\
\bigwedge_{q \in modules} MX^i_q \land MY^i_q
\] (3.2)

This formulation differs from the previous work in [121] in that both the module inputs and outputs are constrained with the simulated values versus just the outputs. This subtle difference results in a strict under-approximation for the abstract instance compared to [121], which is neither an under-approximation or over-approximation. This is due to the fact that in [121] there is no back propagation from the outputs constraints, making the problem less constrained than the original. This leads to the issue of additional spurious solutions that may be found in the abstract problem but not in the original one. However, the formulation in Equation 3.2 overcomes this issue by constraining both the inputs and outputs of a module, resulting in a strict under-approximation as stated in the next lemma.

**Lemma 1** \( InitAbsDebug^k_0 \) is an under-approximation of \( Debug^k_0 \).

**Proof:** Since \( InitAbsDebug^k_0 \) only adds additional constraints, it will result in a subset of the satisfying assignments to \( Debug^k_0 \). Thus, an under-approximation by definition.

Although Equation 3.2 appears to add additional constraints, these constraints in fact allow a simplification of the equation. Applying the module input and output predicates effectively satisfies all the constraints for the corresponding modules. By removing the constraints for each module in the enhanced transition relation, Equation 3.2 can be re-written as:

\[
AbsDebug^k_0 = S^0 \land \Phi_N \land \bigwedge_{i=0}^{k} X^i \land Y^i \land \phi_{glue}(l^i,e) \land \\
\bigwedge_{q \in modules} MX^i_q \land MY^i_q
\] (3.3)

The intuition behind this simplification is that, if the simulated input and output predicates of a module are applied, then the constraints for the module are redundant because the applied

\[^1\text{This may result in a circuit line being constrained twice from both an output of a module and the input of a subsequent module it fans out to. In practice, this does not incur a significant overhead because, when sent to a SAT solver, it will ignore the second identical constraint.}\]
predicates define its behavior. This results in Equation 3.3 returning the same set of solutions as before the simplification, also stated by the following lemma:

**Lemma 2** \( \text{AbsDebug}^k_0 \) will return the same solutions as \( \text{InitAbsDebug}^k_0 \).

**Proof:** \( \text{AbsDebug}^k_0 \) contains the same constraints as \( \text{InitAbsDebug}^k_0 \) except with the module constraints removed. This means that \( \text{AbsDebug}^k_0 \) will return a superset of solutions of \( \text{InitAbsDebug}^k_0 \).

Now we show that any solution found in \( \text{AbsDebug}^k_0 \) will be found in \( \text{InitAbsDebug}^k_0 \). By using the same satisfying assignment as \( \text{AbsDebug}^k_0 \), all the clauses except for the module constraints can be satisfied in \( \text{InitAbsDebug}^k_0 \). However for each module \( q \), if there are no active suspect variables inside the module, then the module constraints \( (M^q_i) \) are fully satisfied because with the applied module input predicate \( (MX^q_i) \) the constraints amount to simulation of the module to generate the values in the module output predicate \( (MY^q_i) \). On the other hand, if there are active suspect variables inside the module, then some of the variables for the corresponding error locations will be disconnected from their fan-in constraints and become free variables. By selecting these free variables to correspond to the simulated values of these nets, the constraints will also be satisfied in a similar manner. In either case, all the module constraints will be satisfied so \( \text{InitAbsDebug}^k_0 \) will always be SAT when \( \text{AbsDebug}^k_0 \) is SAT.

Equation 3.3 gives a strict under-approximation of the debugging instance that is greatly reduced in size by the removal of module constraints. Although it has significantly less flexibility than the original formulation in Equation 2.7, it provides a succinct under-approximation that can be incrementally refined as described in the following subsections.

**Example 14** Using the proposed abstraction technique, the debugging instance from Example 12 is shown in Figure 3.3. Both modules have been replaced with their respective input and output constraints from simulating the erroneous circuit. Notice that the abstract instance is greatly simplified producing an easier problem to solve. In this example, all gates have been removed, no solutions are found and the instance is unsatisfiable.
3.3.2 Module Refinement

The under-approximated abstraction from Equation 3.3 is significantly more constrained than the original debugging formulation in Equation 2.7. The fully abstracted problem will likely find very few solutions because the removed modules prevent the effect of the error models from propagating to the primary outputs. Thus, only suspects that have a direct propagation path to the erroneous output will be found.

The refinement strategy selects a subset of abstracted modules and restores them by removing their module input and output predicates and adding back their original constraints. To select the subset of abstracted modules, an UNSAT core from the solved abstract instance is utilized. Intuitively, the UNSAT core represents the error propagation path backwards from the erroneous output. This path in the concrete instance will pass through many modules. In the abstract one, it will stop at one or more of the abstracted modules.

The refinement strategy uses the UNSAT core to incrementally refine modules on the error propagation path until all modules along the path have been refined. This provides a significant benefit over the refinement strategy from [121], which depends on solutions returned from the abstract problem for refinement. This is due to the fact that the use of solutions for refinement requires a formulation with higher error cardinality because an error can propagate along two separate paths. This is in contrast to using an UNSAT core which will implicitly return multiple
propagation paths if they are responsible for the erroneous output.

To extract relevant information from the UNSAT core, it is traversed and clauses from any abstracted module input or output predicates are found. The modules corresponding to those clauses are marked for refinement in the next iteration. On the other hand, if there are no clauses involving module input or output predicates, then we can safely conclude that we have finished solving the problem.

The intuition behind this termination condition is that since the UNSAT core does not involve any abstracted components, it will also exist in the original debugging formulation from Equation 2.7. Thus, no more solutions would have been found in Equation 2.7 either. Given this condition, a stronger result can be implied that the current refined instance contains enough information to find the same equivalent solutions as the concrete one. This is stated formally in the next theorem.

**Theorem 1** Let $\text{Ref Debug}_0^k$ be the refined instance of $\text{Abs Debug}_0^k$ after zero or more refinement iterations, and $U$ be an UNSAT core generated after blocking all satisfying assignments to solutions for $\text{Ref Debug}_0^k$. If $U$ does not contain any module input or output predicate clauses then the solutions found for $\text{Ref Debug}_0^k$ will be exactly the solutions found for the entire debugging instance $\text{Debug}_0^k$.

**Proof:** Since $\text{Ref Debug}_0^k$ is a refined instance of $\text{Abs Debug}_0^k$, it contains fewer abstracted modules which still results in an under-approximation of $\text{Debug}_0^k$. So by definition it still returns a subset of solutions.

Now we show that any suspect found in $\text{Debug}_0^k$ will be found in $\text{Ref Debug}_0^k$ given the above conditions (i.e., $\text{Ref Debug}_0^k$ will generate a superset of solutions of $\text{Debug}_0^k$). Assume towards a contradiction that $\text{Debug}_0^k$ has a solution, denoted by $V(e)$, that is not found in $\text{Ref Debug}_0^k$ and $U$ is the resulting UNSAT core given above. Let $\text{blocking}(e)$ denote the blocking clauses from $\text{Ref Debug}_0^k$. We know the solutions corresponding to $\text{blocking}(e)$ do not block $V(e)$ from being found in $\text{Debug}_0^k$ or else $\text{Ref Debug}_0^k$ would have found $V(e)$. Therefore, $\text{Debug}_0^k \land V(e) \land \text{blocking}(e)$ is SAT. However, the UNSAT core $U$ only contains clauses from blocked solutions, error trace predicates, glue logic or error cardinality constraints since it does
Figure 3.4: Example of Refinement

not contain any module input or output predicate clauses. This means the UNSAT core $U$ is a subset of the clauses in $\text{Debug}^k_0 \land V(e) \land \text{blocking}(e)$ which is shown to be SAT, leading to a contradiction. So it must be the case that $\text{RefDebug}^k_0 \land V(e)$ is SAT.

Theorem 1 gives us a condition to stop the abstraction refinement process without the need to refine every module. It also shows that this process will eventually provide complete results because in the worst case the abstract problem will degenerate to Equation 2.7 through refinement.

Example 15 Using the proposed abstraction technique, Figure 3.4 shows the debugging instance from Example 12 that has been abstracted and refined once using the proposed technique. Solving the abstract problem returns an UNSAT core involving module $B$ output constraints and the observed erroneous primary output variable $y_2^0$. Following the refinement strategy, module $B$ is refined and the resulting debugging instance is shown in Figure 3.4. This instance can be solved returning a subset of the exact solutions $g_2$ and $g_4$.

Example 16 The combinational circuit from Example 13 can be analyzed with the proposed core-guided abstraction and refinement algorithm. In the first iteration at $N = 1$, the abstract debug problem will be UNSAT with no solutions. This core will involve the outputs of modules $B$, $C$, and $D$, which will be refined in the next iteration. In the subsequent iteration, $\{A\}$ will be found as a solution, as required.
3.3.3 Extension to Basic Scheme

It is possible to provide a more fine-grain refinement strategy compared to the one presented earlier while still maintaining an under-approximated abstraction. Since an UNSAT core simply contains clauses involved in unsatisfiability, the clauses can be mapped back to both their corresponding module and time-frame. Utilizing both pieces of information allows us to refine in a more detailed manner.

This is achieved by traversing the UNSAT core and determining which clauses correspond to added module input and output constraints. For those constraints, we determine both the corresponding module and time-frame instead of just the module. Using this strategy, modules do not need to be refined across all time-frames but only the ones directly involved with the error. This allows us to manage the trace length by only refining modules that are necessary to debug the given problem.

This strategy allows fewer modules across time-frames to be refined but may cause more iterations because only a small amount of the problem is being refined in each iteration. To provide a trade-off between these two metrics, a radius parameter $r$ can be introduced that refines modules that are within $r$ time-frames of the module input or output clause involved in the UNSAT core. This balances the number of iterations against the level of the sequential refinement. Theorem 1 still applies to this sequential refinement strategy as it just refines fewer modules in each iteration compared with the strategy introduced in Section 3.3.2.

Example 17 Figure 3.5 shows the abstract problem from after two iterations of sequential refinement. Notice that only module A in time-frame 1 and module B in time-frame 2 are refined since their module input and output constraints were the only ones in the UNSAT core after solving each iteration. Solving this refined instance produces results $g_0$, $g_2$, $g_3$, and $g_4$, that is, the exact set of solutions. After solving this instance, we can get an UNSAT core with a path from primary input variables $x_0^1$, $x_1^1$ and $x_2^0$ to primary output variables $y_0^2$. According to Theorem 1, this results in a complete set of solutions.

3.3.4 Overall Algorithm
Algorithm 3.1 Core-Guided Abstraction and Refinement

1: \( N := \) error cardinality
2: \( e := \) set of potential suspect variables
3: \( k := \) length of error trace
4: \( sols := \) solutions found by algorithm
5: \textbf{procedure} AbstractionRefinement
6: \quad \textit{inst} \leftarrow \textit{AbsDebug}^0_{k-1}(N,e)
7: \quad \textbf{while} true \textbf{ do}
8: \quad \quad \textit{sols} \leftarrow \textit{sols} \cup \text{SolveAll}(\textit{inst})
9: \quad \quad \textit{U} \leftarrow \text{ExtractUNSATCore}(\textit{inst}, \textit{sols})
10: \quad \quad \textbf{if} \ U \cap (MX \cup MY) = \emptyset \textbf{ then}
11: \quad \quad \quad \textbf{return} \ \textit{sols}
12: \quad \quad \textbf{end if}
13: \quad \quad \textit{inst} \leftarrow \text{Refine}(\textit{inst}, \textit{U})
14: \quad \textbf{end while}
15: \textbf{end procedure}
Figure 3.5: Example of Sequential Refinement

The overall abstraction and refinement procedure is found in Algorithm 3.1. It begins by creating an abstract debugging instance on line 6 as described by Equation 3.3. Then, it iteratively solves the abstract problem and refines it in the loop of lines 7-14. After each abstract problem is solved, the UNSAT core is extracted and analyzed to determine if it involves any module input or output constraints (lines 8-10). If no constraints are found, then it terminates, returning an exact set of solutions according to Theorem 1 (line 11). Otherwise, the problem is refined according to one of the refinement strategies presented in Section 3.3.2 or Section 3.3.3. Lemma 1 is implicitly used when creating the refined problem on line 13. This step prunes previously found solutions from being considered in future iterations since we know that each abstract instance is an under-approximation.

The algorithm will always terminate because in the worst case the refinement strategy will cause the abstract problem to degenerate to Equation 2.7 resulting in a true condition in line 10.

3.4 Experiments

This section presents the experimental results for the proposed abstraction and refinement algorithm. All experiments are run using a single core of a Core 2 quad-core 2.66 Ghz workstation with 8 GB of RAM and a timeout of 3600 seconds. Algorithm 3.1 is implemented for both refinement strategies and results are compared to a sequential SAT-based debugger [128] and
Table 3.1: Design Statistics

<table>
<thead>
<tr>
<th>Design</th>
<th># gates</th>
<th># flipflops</th>
<th># suspect locations</th>
<th># modules</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac97ctrl</td>
<td>30540</td>
<td>2482</td>
<td>728</td>
<td>28</td>
</tr>
<tr>
<td>div64bits</td>
<td>80907</td>
<td>16910</td>
<td>114</td>
<td>2</td>
</tr>
<tr>
<td>fdct</td>
<td>394133</td>
<td>5717</td>
<td>4387</td>
<td>3</td>
</tr>
<tr>
<td>fpu</td>
<td>83303</td>
<td>761</td>
<td>536</td>
<td>8</td>
</tr>
<tr>
<td>fxu</td>
<td>676928</td>
<td>29080</td>
<td>17418</td>
<td>13</td>
</tr>
<tr>
<td>memctrl</td>
<td>55689</td>
<td>1361</td>
<td>2040</td>
<td>10</td>
</tr>
<tr>
<td>rsdecoder</td>
<td>16072</td>
<td>526</td>
<td>1100</td>
<td>7</td>
</tr>
<tr>
<td>comm</td>
<td>1028772</td>
<td>33382</td>
<td>27176</td>
<td>9</td>
</tr>
<tr>
<td>vga</td>
<td>175999</td>
<td>17168</td>
<td>902</td>
<td>7</td>
</tr>
<tr>
<td>conmax</td>
<td>140892</td>
<td>1106</td>
<td>891</td>
<td>26</td>
</tr>
</tbody>
</table>

The previous work in abstraction and refinement for design debugging \cite{121}. PicoSAT-v913 \cite{18} is used to solve all SAT instances and generate all UNSAT cores. A commercial Verilog front-end \cite{139} is used to parse and elaborate all designs to derive a gate-level netlist and RTL-to-gate mapping.

The effectiveness of the proposed algorithm is shown on industrial Verilog RTL designs from OpenCores \cite{111} and two commercial designs (fxu, comm) provided by our industrial partners. Table 3.1 shows the statistics for each design: the design name, number of synthesized gates, number of state elements, number of initial potential suspect locations and the number of modules available for abstraction. The designs in this table may be used in experiments for other chapters, but may differ in their design characteristics. This difference can attributed to the version of the design and the version of the Verilog frontend.

Each debugging instance is created by randomly selecting a line in the RTL and inserting a typical industrial RTL error such as a wrong state transition, incorrect operator or incorrect module instantiation. The erroneous design is then run through an industrial simulator \cite{101} with the accompanying testbench where a failure is detected and the error trace recorded. Suspects returned by the debuggers correspond to lines in the RTL such as assignments, if statements, instantiations etc. It should be emphasized that a single location in the RTL typically corresponds to multiple locations in a gate-level formulation. All abstraction techniques use modules which correspond to top-level instantiations in the Verilog design.

The results for all experiments are shown in Table 3.2. Each design is used to create an
instance by inserting an error into the design. A number is appended to each design name to indicate which error is inserted and it is shown in the first column of Table 3.2. Four sets of experiments are run on each instance corresponding to each of the different debugging methods used. The first set uses the technique based on [128] which we denote as SAT-based debugging. The second set is run with the previous work from [121] which we denote as Suspect Refinement. The final two sets correspond to the abstraction and refinement procedure presented in Algorithm 3.1 with the two refinement strategies from Section 3.3.2 and Section 3.3.3 which we will denote as Module Refinement and Sequential Refinement respectively. Sequential refinement is used with a radius parameter of $r = 20$.

The latter 12 columns in Table 3.2 show the run-time, peak memory and number of solutions returned by each of the four sets of experiments. A * beside the number of solutions denotes that the actual inserted error was found. A TO (MO) entry in the table is used to refer to a time-out (memory-out) condition for that experiment. For TO or MO conditions, the solutions column lists the number of partial solutions returned by that method before the condition occurs.

The benefit of the proposed abstraction and refinement technique is apparent when comparing the number of instances that return solutions. SAT-based debugging is only able to return solutions for 9, or 41%, of the 22 instances while Module Refinement and Sequential Refinement are able to return solutions for 19 and 22 of the instances respectively. The excessive size of the unrolled debugging problem from Equation 2.7 prevents the SAT-based debugging formulation from fitting into memory while the proposed algorithm is able to deal with these instances.

Comparing the total number of solutions, we can also see the benefit of the proposed technique compared to Suspect Refinement. In almost all of the instances, Suspect Refinement returns an excessive number of solutions because it must increase its error cardinality. This results in a superset of the solutions from Equation 2.7. This is a significant benefit of Module and Sequential Refinement techniques over Suspect Refinement because an excessive number of solutions negates the benefit of automated debugging techniques to narrow down the list of potential suspects. The aggressive abstraction from the Suspect Refinement technique combined with the need for the refinement strategy to increase the error cardinality are the primary
reasons for the large increase in solutions. The refinement strategy also leads to excessive run-times where 14 of the 22 instances cause a time-out due to increased error cardinality. In these cases, the error cardinality is increased in an early iteration before many of the modules are refined, hence a lower reported peak memory for Suspect Refinement.

For the 7 instances where SAT-based debugging does not cause a memory-out or time-out, the run-time performs better in some cases such as fxu2 and worse in others (fxu1). On average, Module Refinement and Sequential Refinement had a 2.5% and 187.2% increase in run-time, respectively. Using a geometric mean, the two respective refinement strategies had a 1.03x and 0.45x speed-up on average. Abstraction and refinement aims to target large industrial instances, so on these small instances where SAT-based debugging can complete the trade-off is less beneficial.

For memory we see similar mixed results for the same 7 instances. On average, Module Refinement and Sequential Refinement had a 5.5% and 13.0% decrease in peak memory, respectively. Using a geometric mean, the two respective refinement strategies used 0.91x and 0.84x of peak memory compared with SAT-based debugging, on average. In the case of fxu3, Theorem 1 is applied generating complete results where many of the modules did not need to be refined. This results in a dramatic reduction in peak memory from 7390 MB with SAT-based debugging to 3372 MB and 4181 MB with the Module and Sequential Refinement techniques, respectively. In other cases where refinement required all the modules to be restored, such as fxu2, the peak memory is approximately the same (7813 MB vs. 7907 MB and 7970 MB). This slight increase is primarily due to the overhead for abstraction and refinement and the enabling of tracing UNSAT cores in the SAT solver. However, despite this slight overhead in memory, the majority of the instances still run using the proposed algorithm where it would not be possible otherwise.

With the two proposed refinement strategies, Theorem 1 is successfully applied in one instance for Module Refinement (fxu3) and four instances for Sequential Refinement (fpu2, fxu3, memctrl12, vga3) where the algorithm terminates without refining all modules. Since this theorem relies on UNSAT cores, it is highly dependent on what the solver returns. Sequential Refinement typically requires more iterations, therefore it has a higher chance that it will find
an UNSAT core that can be applied with Theorem \[ \text{mod} \] However, for both these refinement strategies a dramatic reduction in run-time and memory is achieved when applicable.

Figure 3.6 shows the number of solutions returned from the abstract problem per iteration for both the proposed refinement strategies for \text{ac97ctrl1}, \text{fpu1}, \text{fxu2} and \text{conmax1} where \text{mod} stands for Module Refinement and \text{seq} stands for Sequential Refinement. These instances were chosen to illustrate the differences between Module and Sequential refinement. The first iteration for all instances is the same regardless of the refinement strategy because the abstract problems are identical. Benchmark \text{ac97ctrl1} shows a case where the refinement of modules across all time-frames is beneficial. In the second iteration of this instance, Module Refinement finds 11 solutions versus 4 for Sequential Refinement. This demonstrates that many of the suspects require propagation across many time-frames, a disadvantage for Sequential Refinement which takes many iterations to achieve the same result. Cases \text{fpu1} and \text{conmax1} on the other hand, show great benefits from using Sequential Refinement. In these cases, the refinement of modules across all time-frames causes the instance to become too large to fit into memory causing a memory-out condition for Module Refinement. This is not an issue for Sequential Refinement which completes more iterations and finds more solutions.
3.5 Summary

This chapter presents a novel core-guided abstraction and refinement algorithm for design debugging to manage the design size component of debugging complexity. This is accomplished by creating an under-approximate abstract problem that uses simulated values to replace modules of the original design. Following this, UNSAT cores are used to direct which modules to refine. An extension of the basic refinement strategy utilizes the UNSAT core to refine modules across time-frames as well. Experimental results on industrial designs show major benefits in run-time and peak-memory compared to previous state-of-the-art techniques.
Table 3.2: Abstraction and refinement experiments for the proposed technique. Compares SAT-based debugging \cite{128} and previous work in abstraction and refinement for design debugging (denoted as suspect refinement) \cite{121} to the two refinement schemes proposed in Section 3.3.2 and Section 3.3.3, which are denoted as module refinement and sequential refinement, respectively. A * beside the number of solutions denotes that the actual inserted error was found.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time (s)</td>
<td>peak memory (MB)</td>
<td># solutions</td>
<td>time (s)</td>
</tr>
<tr>
<td>ac97ctrl1</td>
<td>679</td>
<td>7560</td>
<td>22*</td>
<td>TO</td>
</tr>
<tr>
<td>div64bits1</td>
<td>234</td>
<td>6428</td>
<td>5*</td>
<td>302</td>
</tr>
<tr>
<td>div64bits2</td>
<td>220</td>
<td>6240</td>
<td>5*</td>
<td>296</td>
</tr>
<tr>
<td>div64bits3</td>
<td>227</td>
<td>6333</td>
<td>5*</td>
<td>298</td>
</tr>
<tr>
<td>fdct1</td>
<td>N/A</td>
<td>MO</td>
<td>0</td>
<td>TO</td>
</tr>
<tr>
<td>fdct2</td>
<td>N/A</td>
<td>MO</td>
<td>0</td>
<td>TO</td>
</tr>
<tr>
<td>fpu1</td>
<td>N/A</td>
<td>MO</td>
<td>0</td>
<td>TO</td>
</tr>
<tr>
<td>fpu2</td>
<td>N/A</td>
<td>MO</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>fxu1</td>
<td>1059</td>
<td>7219</td>
<td>24*</td>
<td>TO</td>
</tr>
<tr>
<td>fxu2</td>
<td>560</td>
<td>7813</td>
<td>18*</td>
<td>TO</td>
</tr>
<tr>
<td>fxu3</td>
<td>267</td>
<td>7390</td>
<td>1*</td>
<td>TO</td>
</tr>
<tr>
<td>memctrl1</td>
<td>N/A</td>
<td>MO</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>memctrl2</td>
<td>N/A</td>
<td>MO</td>
<td>0</td>
<td>TO</td>
</tr>
<tr>
<td>rsdecoder1</td>
<td>TO</td>
<td>1645</td>
<td>101</td>
<td>TO</td>
</tr>
<tr>
<td>rsdecoder2</td>
<td>TO</td>
<td>2021</td>
<td>38</td>
<td>TO</td>
</tr>
<tr>
<td>comm1</td>
<td>N/A</td>
<td>MO</td>
<td>0</td>
<td>TO</td>
</tr>
<tr>
<td>comm2</td>
<td>N/A</td>
<td>MO</td>
<td>0</td>
<td>TO</td>
</tr>
<tr>
<td>vga1</td>
<td>N/A</td>
<td>MO</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>vga2</td>
<td>N/A</td>
<td>MO</td>
<td>0</td>
<td>TO</td>
</tr>
<tr>
<td>vga3</td>
<td>N/A</td>
<td>MO</td>
<td>0</td>
<td>TO</td>
</tr>
<tr>
<td>conmax1</td>
<td>N/A</td>
<td>MO</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>conmax2</td>
<td>N/A</td>
<td>MO</td>
<td>0</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Chapter 4

Path Directed Abstraction and Refinement

4.1 Introduction

Contemporary automated design debugging techniques [55, 119, 128] make extensive use of formal methods and time-frame expansion to model the sequential behavior of a design. For current industrial design sizes and error trace lengths, the complexity of the debugging problem may increase dramatically leading to performance issues, rendering these techniques sometimes impractical. Tackling these two major components of debugging complexity (i.e., design size and error trace length) has been the focus of recent research efforts. The previous chapter focused on the former component of debugging complexity, this chapter focuses on the latter.

Abstraction and refinement techniques [83, 121], as described in the previous chapter, have been developed to overcome the design size aspect of debugging complexity. These methods perform analysis on an abstract model of the design that is significantly smaller than the concrete one. Subsequently, they iteratively refine the abstraction to determine a sufficient set of components to solve the debugging problem. Orthogonally, time-windowing [82, 147] aims to deal with the excessive error trace length aspect of the debugging complexity. Instead of analyzing the entire error trace at once, they iteratively analyze a sliding window of time-frames along its length. To properly model the propagation of the error, time-frames not included in
the current window are approximated using various methods. The use of approximation greatly reduces the memory requirements of the debugger but can also lead to poor resolution. This translates to the algorithm marking too many additional components as potentially erroneous, negating the benefits of localization in the first place.

In an effort to overcome this limitation, this chapter introduces a novel abstraction and refinement algorithm for design debugging built upon a time-windowing framework to manage excessive error trace lengths. The key innovation of the algorithm is a path-directed abstraction that conceptually represents relevant structural circuit paths in the time-frame expanded circuit. This abstraction provides a concise approximation of non-modeled time-frames while simultaneously providing an efficient representation for refinement, which consists of adding back omitted structural circuit paths. The result is an algorithm that dramatically reduces the memory requirements of debugging, while mitigating the incomplete results of past time-windowing techniques.

Due to the inherent iterative nature of the algorithm, performance remains the crucial issue behind its practical implementation. This chapter additionally introduces several key optimizations to the basic refinement strategy that significantly reduce the number of iterations needed to achieve complete results. However, in certain worst case scenarios, the algorithm may still suffer from an excessive number of refinement steps. To mitigate this worst case scenario, a flexible path-directed algorithm is presented that allows a trade-off between performance and resolution by limiting the amount of refinement steps per problem instance. This is shown experimentally to dramatically reduce the run-time requirements while having minor effects on the final resolution.

An extensive set of experimental results on large hardware designs from OpenCores\[111\] and industrial partners illustrates the benefits of this work. The proposed approach can analyze error traces that are 64.6% longer while simultaneously using significantly less memory when compared to previous work. Additionally, the refinement strategy of the flexible algorithm is able generate only 1.5% additional spurious solutions compared to 114.7% when no refinement is used.

The remaining sections are as follows. Section\[4.2\] presents background material. Section\[4.3\]
and Section 4.3 describe the proposed approach and refinement improvements, respectively. Section 4.5 presents a new flexible algorithm that allows a trade-off between performance and quality of results while Section 4.6 presents experimental results. Finally, Section 4.7 concludes this work.

4.2 Preliminaries

4.2.1 Time Diagnosis and Time-Windowing

An alternate formulation of the debugging problem, known as time diagnosis [142], can be generated by using a variation of Equation 2.7 (from Section 2.6). The key difference is that when unrolling the enhanced transition relationship, the suspect variable for a component is shared only within a fixed time-window of clock cycles rather than the entire error trace. That is, each component has several associated suspect variables, where each variable corresponds to a different time-window. This models the excitation of an error within a time-window, but not across time-windows. This formulation has been shown [142] to dramatically reduce problem complexity while still being able to model real-life bugs with long error traces. Due to its relation to this work, we provide more detail on the underlying formulation.

To model a single time-window Equation 2.7 can be used. However, as previously mentioned, if the observed failure is not within this window, the instance will be trivially satisfiable. To remedy this issue, one can model subsequent suffix time-windows so that the observed failure at the primary outputs is included. For time-windows of width $w$, we can model the $p^{th}$ window in the sequence by the following equation:

$$W_p = \bigwedge_{i=p-w}^{[p+1)-w-1} (X^i \land Y^i \land T^i)$$

(4.1)

Note that if $N = 0$ is set in Equation 2.7 it simplifies to a time-frame shifted version of Equation 4.1 (with the addition of the initial state constraints) because it eliminates all additional circuitry relating to error models.

By combining Equation 2.7 and 4.1 one can overcome the issue of debugging a time-window that does not contain the observed failure. This formulation is shown with respect to a set of
time-windows from \( p \) to \( q \) with a width of \( w \):

\[
\text{TimeDebug}_p^q(N) = \text{Debug}_{[p+1,w]-1}(N) \land \bigwedge_{i=p+1}^{q} W_i
\]  

(4.2)

This formulation permits the effects of the error models in the time-window under analysis (e.g., \( p \)) to propagate and affect the observed failure in a subsequent suffix time-window (e.g., \( q \)).

**Example 18** Figure 4.1(a) visualizes Equation 4.2 when \( p = q = 1 \), \( w = 1 \) and \( N = 1 \). The circuit under debug has three internal gates \((g_1, g_2, g_3)\), one input \((x_0)\), one output \((y_0)\), and three state-variables \((s_0, s_1, s_2)\). In this circuit, \( g_1 \) and \( g_2 \) form module \( A \) and \( g_3 \) forms module \( B \). The error models are denoted by \( \otimes \). There are two suspect variables, \( e_a, e_b \) corresponding to modules \( A \) and \( B \) respectively. The suspect variable \( e_a \), corresponds to the two outputs of \( g_1 \) and \( g_2 \) from module \( A \). Similar is the situation for suspect variable \( e_b \) and gate \( g_3 \). Error trace values for the initial state, input and expected outputs are shown in the figure.

When the instance in Figure 4.1(a) is passed to a SAT-solver, one solution is returned, namely \( e_b = 1 \), that corresponds to module \( B \) being a suspect. When \( \overline{e_b} \) is added as a unit clause to block it from appearing again, the instance is \text{UNSAT} indicating that no more solutions are found.

Figure 4.1(b) shows a visualization of Equation 4.2 with the next time-window where \( p = 0, q = 1, w = 1 \) and \( N = 1 \). In this instance, the suspect variables now only correspond to modules in the earliest time-frame. When sent to the SAT-solver, one solution is returned where \( e_a = 1 \). After blocking it as a solution, the instance is \text{UNSAT} indicating that all solutions have been found.

Time-windowing techniques \cite{82} use a similar formulation to Equation 4.2. For a given time-window width, they divide the error trace into multiple non-overlapping time-windows. Each window is iteratively analyzed starting from the last time-window in the error trace that contains the observed failure (i.e., \( p = q \) for Equation 4.2). This “sliding window” of time-frames moves towards earlier time-windows \((p = q, p = q - 1, \ldots, p = 0)\) iteratively analyzing the error trace. To ensure that the current time-window under analysis (e.g., window \( p \)) can
propagate the effect of its error models to the observed failure (e.g., window q), various methods are used to model the suffix time-windows.

In the most basic formulation \[82\], suffix time-windows in the trace are explicitly modeled as in Equation 4.2. This results in iteratively solving instances of Equation 4.2 where each iteration results in a larger problem size. The proposed strategy leads to complete results but suffers from performance and memory issues in later iterations. Subsequent formulations forego the need to explicitly model suffix time-frames and instead rely on approximations, such as interpolants \[82\], to model later time-windows \((W_i)\). This approximation is essentially a formula that is implied by \((W_i)\), resulting in an over-approximation for the entire debug time-
windowing instance. This ensures that the effect of the error models in the window under analysis can propagate and affect the observed failure. The approximations effectively reduce the problem size but at the cost of additional spurious solutions that would not otherwise be in the concretely modeled problem. With no means of refinement, these previous techniques may suffer from poor quality of results.

4.2.2 Counter-Example Guided Abstraction and Refinement

SAT-based Counter-Example Guided Abstraction and Refinement (CEGAR) [36] is an iterative model checking algorithm composed of three general steps: abstraction, model checking and refinement. The pseudo-code for the algorithm is shown in Algorithm 4.1. The initial abstraction begins with an over-approximation of the concrete problem (line 2). During the model checking phase, the current abstract instance is model checked on line 3 and line 9. If no counter-example is found, then the procedure has proved the property due to the over-approximate abstraction and returns on line 11. Otherwise the counter-example may be spurious and must be verified. This is done by applying the counter-example to an unrolled concrete transition relationship using the IsValidCounterExample procedure. If this instance is SAT then the counter-example is valid, and a bug has been found. In the UNSAT case, the counter-example is spurious and refinement begins which strengthens the abstraction to ensure that the spurious counter-example is not found (line 8). After refinement, this process repeats until either a valid counter-example is found, or the property is determined to be valid.

4.3 Path Directed Abstraction and Refinement

This section presents an abstraction and refinement algorithm for design debugging built upon both a CEGAR [36] and time-windowing framework with the debugging model from Equation 4.2.

This key idea is shown in Figure 4.2 with an error trace containing $k+1$ time-frames. In this example, the trace is divided into $\frac{k+1}{w}$ non-overlapping time-windows with $w = 2$. As with other time-windowing techniques, the algorithm iteratively analyzes non-overlapping time-windows
Algorithm 4.1 Counter-Example Guided Abstraction and Refinement

1: procedure CEGAR

2: \quad \text{inst} \leftarrow \text{GenerateInitialAbstraction}

3: \quad \text{cex} \leftarrow \text{ModelCheck} (\text{inst})

4: \quad \text{while } \text{cex} \neq \emptyset \text{ do}

5: \quad \quad \text{if } \text{IsValidCounterExample} (\text{cex}) \text{ then}

6: \quad \quad \quad \text{return } \text{cex}

7: \quad \quad \text{end if}

8: \quad \quad \text{inst} \leftarrow \text{Refine} (\text{inst}, \text{cex})

9: \quad \quad \text{cex} \leftarrow \text{ModelCheck} (\text{inst})

10: \quad \text{end while}

11: \quad \text{return } \emptyset

12: \text{end procedure}

starting from the latest window. The abstract time diagnosis instance for time-window \( \frac{i}{w} \) is shown in greater detail. This instance is constructed by concretely modeling the current window (e.g., cycles \( i \) and \( i + 1 \)) and abstracting all later time-windows in the suffix (e.g., cycles \( i + 2 \) to \( k \)). This abstraction is iteratively refined in a CEGAR-like abstraction and refinement loop with three overall steps:

1. **Abstraction**: Generate an abstract time diagnosis instance by concretely modeling the current time-window combined with an over-approximate abstraction for the later suffix time-windows. This abstraction is conceptually based upon structural circuit paths that directly led to the observed failure in the error trace.

2. **Debugging**: Find all solutions to the abstract time diagnosis debugging instance. Since this instance over-approximates Equation 4.2, it generates a superset of the solutions compared to explicit modeling. To ensure only valid solutions are returned, each solution must then be verified in the refinement phase.

3. **Refinement**: For each solution, propagate its associated satisfying assignment forward to concretely modeled time-windows in separate SAT instances. If each instance is SAT, then
the solution is valid and can be blocked from appearing in future iterations. Otherwise, a
conflict is generated indicating that the abstraction is not sufficient to prove the current
solution. In this case, refinement occurs where additional paths (in the form of clauses
involved in the conflict graph) are added back to the abstraction.

4. Repeat steps (2)-(3) until no more new valid solutions are found in step (2).

After all solutions have been found for the current time-window, the algorithm moves on to
an earlier time-window re-using the refined abstraction generated for suffix time-windows from
the current iteration. The following example gives a high-level intuition on how the overall
algorithm operates.

**Example 19** Figure 4.3 shows a high-level example of how the overall algorithm works. In this
example, there are four time-frames with \( w = 2 \). The initial window \( p = q = 1 \) is analyzed using
a standard suffix time-diagnosis debugging formulation (step 1). After all solutions have been
found, the UNSAT instance can be used to extract a conflict graph and its associated clauses,
which defines the initial abstraction for this time-window (step 2). This intuitively represents a
path in the unrolled circuit. Next, time-window 0 is analyzed using the initial abstraction (step
3). This instance finds a single solution \( A \). To verify that this is a valid solution, its satisfying
assignment is propagated forward to a concrete instance of time-window 1 (step 4). In this
case, \( A \) is not valid resulting in an UNSAT instance, which can be used to refine the abstraction
(step 5). Using this refined abstraction, time-window 0 is analyzed again and finds solution \( B \)
(step 6). As before, \( B \) is verified by propagating its assignment forward, and in this case it is a
valid solution, which is subsequently blocked (step 7). Finally, time-window 0 is analyzed again
to find additional solutions, but returns UNSAT (step 8). This means that no more solutions
exist and analysis on this window is complete.

The following subsections describe the formulation, theoretical results and pseudo-code for
the path-directed abstraction and refinement algorithm in greater detail.
4.3.1 Path-based Abstraction

The path-based abstraction approximates Equation 4.2 by replacing consecutive windows of time-frames \( W_i \) with an abstract version, denoted by \( AbsW_i \). This is initially formulated by adding paths in the time-frame expanded circuit that are directly involved in the observed failure. Practically, this is a set of clauses extracted from the SAT-solver conflict graph generated while modeling the observed failure. The rest of this subsection describes this process in greater detail.

The first step in the algorithm analyzes time-window \( p = q \) using Equation 4.2. In this initial iteration, no abstract time-windows are needed because the failure can be observed in the current window. However, once this window has finished its analysis, we must generate the first abstract time-window \( (AbsW_q) \) for cycles \( q \cdot w \) to \( [(q + 1) \cdot w] - 1 \). To construct \( AbsW_q \), first notice that by setting \( N = 0 \) in Equation 4.2 it simplifies to \( W_q \) with the addition of the initial state constraints. This precisely models the circuit behavior for cycles \( q \cdot w \) to \( [(q + 1) \cdot w] - 1 \) that led to the observed failure, which will result in an UNSAT instance. Thus, the resulting conflict graph from the SAT-solver will contain clauses that are directly involved in the observed failure at the primary outputs. These clauses, excluding the initial state constraints, form the
initial abstraction $AbsW_q$.

In general, the abstraction is formed by maintaining a set of clauses for each time-window that has been previously analyzed. When a time-window has completed analysis, we set $N = 0$ for that instance which is guaranteed to be UNSAT (see Section 2.6). From this UNSAT instance, we extract clauses within the current window that are directly involved in the conflict.

This is shown in the following equation:

$$AbsW_i = Conflict_i(AbsDebug^a_i(N = 0)) \quad (4.3)$$

Here, $Conflict_i$ denotes all clauses with variables within time-frames $i \cdot w$ to $(i + 1) \cdot w - 1$ involved in the conflict of the input formula, excluding the initial state constraints. This can be extracted by analyzing the conflict graph of the SAT-solver when it returns UNSAT, extracting only clauses that correspond to the relevant time-frames. $AbsDebug^a_i(N)$ denotes the abstract debugging instance for the previously analyzed time-window. In the base case for $i = q$, this
simplifies to Equation 2.7.

Using Equation 4.3, the initial abstract debugging problem can be created, shown in the following formula:

\[
\text{AbsDebug}_{p}^{q}(N) = \text{Debug}_{p+w}^{[(p+1)-w]-1}(N) \land \bigwedge_{i=p+1}^{q} \text{AbsW}_{i}
\]  

(4.4)

This formula mirrors Equation 4.2 except it uses the abstract windows for later time-frames instead of explicitly modeling them. It results in a greatly reduced memory footprint because the abstract windows are typically much smaller than the explicit model. In addition, since each abstract window \(\text{AbsW}_{i}\) contains either clauses that were in the explicit model of that window \(W_{i}\) or implied by it, Equation 4.4 is in fact an over-approximation of Equation 4.2 as stated in the next lemma.

**Lemma 3** Let \(\mathcal{E}\) be a set of \(N\) active suspect variables found in a satisfying assignment to \(\text{TimeDebug}_{p}^{q}(N)\), then there is a satisfying assignment to \(\text{AbsDebug}_{p}^{q}(N)\) that will also contain the active suspect variables from \(\mathcal{E}\).

**Proof:** Let \(\mathcal{A}\) be the satisfying assignment to \(\text{TimeDebug}_{p}^{q}(N)\). We will show that each of the sub-formulas of \(\text{AbsDebug}_{p}^{q}(N)\) is satisfiable under \(\mathcal{A}\). \(\text{Debug}_{p+w}^{[(p+1)-w]-1}(N)\) is SAT under \(\mathcal{A}\) because it is a subset of the clauses in \(\text{TimeDebug}_{p}^{q}(N)\). Each \(\text{AbsW}_{i}\) is derived from extracting clauses from a SAT-solver conflict graph that are within the same time-frames modeled by \(W_{i}\). This means that these clauses are either a subset of \(W_{i}\) or implied by the overall problem. Therefore, if all \(W_{i}\) are SAT under \(\mathcal{A}\), so are all \(\text{AbsW}_{i}\). Since each component is SAT under \(\mathcal{A}\), so is \(\text{AbsDebug}_{p}^{q}(N)\) with active suspect variables \(\mathcal{E}\).

Lemma 3 implies that the initial abstract instance will return a superset of debugging solutions when compared to explicit modeling. This abstraction resembles the method for approximating time-frames in [82] because they both over-approximate a suffix of time-frames and are derived from an UNSAT problem. However, they differ in the method used to compute the approximation. While [82] calculated an interpolant directly from the proof of unsatisfiability, our method uses the conflict graph. This method is more efficient because it removes the need to record and traverse the UNSAT proof to calculate the interpolant, while still retaining
equivalent information. Additionally, the abstraction allows an efficient means to incrementally refine the problem, which will be described in later sections.

Due to the abstraction providing a superset of the debugging solutions, each solution must be verified to ensure it is not spurious. To accomplish this, the satisfying assignment of each solution is propagated forward to each subsequent concretely modeled time-window. If a solution is indeed spurious, the resulting conflict will act as a refinement step, a process discussed in the following subsection.

**Example 20** Consider the debugging instance from Figure 4.1(a) in Example 18. Once this instance returns UNSAT, the algorithm sets $N = 0$, it runs a SAT-solver and it extracts clauses involved in the resulting conflict graph to generate $\text{AbsW}_1$.

In the next iteration of the main loop, it generates an abstract debugging problem using Equation 4.4 with $p = 0, q = 1, w = 1$ and $N = 1$. A visualization of this instance is found in Figure 4.4. Here, most of circuitry from time-frame 1 has been abstracted leaving only the path directly involved in the conflict from the SAT-solver. Note this path is not unique and another conflict graph has the potential to generate a different path.
4.3.2 Path Directed Refinement

Due to the use of abstract windows in Equation 4.4, a satisfying assignment, denoted by \( \mathcal{A} \), may not correspond to one in the concrete instance and therefore may be spurious. A straightforward method to verify \( \mathcal{A} \) would be to apply it to the entire concrete instance in Equation 4.2. However, this would negate the benefit of a reduced memory footprint since the instance would involve explicit modeling of all time-frames. Instead, by utilizing the previous abstract windows, it is possible to create several smaller SAT instances that can equivalently verify \( \mathcal{A} \).

In order to propagate \( \mathcal{A} \) forward, only a subset of assignments are actually needed. Notice that in Equation 4.4, time-frames from \( p \cdot w \) to \( [(p + 1) \cdot w] - 1 \) are modeled explicitly, while the remaining are not. This means that an assignment for the first \( p \cdot w \) to \( [(p + 1) \cdot w] - 1 \) time-frames on the abstract model should also work for the respective frames on the concrete model in Equation 4.2. However, the only way for these concrete time-frames to affect forward time-frames are through the state variables at time-frame \( (p + 1) \cdot w \). If these state assignments are used to constrain subsequent concretely modeled time-frames, then we can iteratively propagate the effect of the original assignment forward to each concretely modeled time-window. We denote this subset of assignments of \( \mathcal{A} \) on state variables for time-frame \( i \) by \( \text{cube}^i \).

To accomplish this propagation, we create multiple instances each of which models precisely \( w \) concrete time-frames and uses the abstract windows for the others. This construction is represented by the following equation:

\[
\text{Prop}^q_{r+1} = \text{cube}^{r \cdot w} \land W_r \land \bigwedge_{i=r+1}^{q} \text{Abs} W_i
\]  

(4.5)

The state assignment \( \text{cube}^{(p+1) \cdot w} \), extracted from \( \mathcal{A} \), is propagated using the above equation to generate \( \text{Prop}^q_{p+1} \). If this results in \( \text{SAT} \), another cube, \( \text{cube}^{(p+2) \cdot w} \), will be extracted and propagated to the next instance, \( \text{Prop}^q_{p+2} \), and so on, until all time-frames have been verified. If all subsequent instances result in \( \text{SAT} \), then the original abstract satisfying assignment can be extended to one in the concrete model. This is stated more precisely in the following lemma.

**Lemma 4** Let \( \mathcal{E} \) be a set of \( N \) active suspect variables found in a satisfying assignment of
$\text{AbsDebug}_p^q(N)$. If $\text{AbsDebug}_p^q(N)$, $\text{Prop}_p^{q+1}$, $\ldots$, $\text{Prop}_p^q$ are SAT, then there is a satisfying assignment to $\text{TimeDebug}_p^q(N)$ that will also contain the active suspect variables from $E$.

**Proof:** Let $A_p, A_{p+1}, \ldots, A_q$ be the satisfying assignments to $\text{AbsDebug}_p^q(N)$, $\text{Prop}_p^{q+1}$, $\ldots$, $\text{Prop}_p^q$, respectively. We show how to construct an assignment $A$ to $\text{TimeDebug}_p^q(N)$ from $A_p, A_{p+1}, \ldots, A_q$.

From $A_p$, we add all assignments to variables involved in the subformula $\text{Debug}_{([p+1]w)-1}(N)$ to $A$. From each of the subsequent $A_r$, we add assignments to all variables of $W_r$ from the respective instance $\text{Prop}_r^q$. Notice the overlapping variables that were added to $A$ are precisely the state variable from the cubes $\text{cube}^{p+w}$, $\text{cube}^{(p+1)-w}$, $\ldots$, $\text{cube}^{q-w}$. But from the formulation of $\text{Prop}_r^{q-1}$, each cube (i.e., $\text{cube}^{r-w}$) is generated from an instance that is constrained by the previous cube, $\text{cube}^{(r-1)-w}$, except for the first one derived from the assignment $A_p$. This means that there is no conflict between these overlapping state variables because each one is implied by the previous one in the sequence which originates from $A_p$.

The final constructed assignment $A$ composes a full assignment to $\text{TimeDebug}_p^q(N)$ since it contains assignments to all variables in each component of the formula. Moreover, $A$ is a valid satisfying assignment to each of the components of $\text{TimeDebug}_p^q(N)$. We can see this because we constructed $A$ by extracting exactly the assignments involved in the concrete parts of the abstract formulas which are identical to the concrete model. These concrete parts compose exactly the clauses of $\text{TimeDebug}_p^q(N)$, so it too is SAT under $A$ with active suspect variables $E$.

After a set of active suspect variables is found, they are blocked so that the next solution can be found for $\text{AbsDebug}_p^q$. Notice that the propagation generates a separate instance for every $w$ time-frames resulting in exactly $w$ concretely modeled time-frames for any given instance. This key point ensures that the memory footprint of the entire process is kept to a minimum.

On the other hand, if $\text{Prop}_p^q$ is UNSAT, it means that the original assignment $A$ cannot be extended to the concrete model. In this case, the abstract window was not sufficiently refined and additional clauses must be added. Similar to the initial abstraction, we extract clauses from the SAT-solver conflict graph and add them to the abstract window. Intuitively, this indicates
that additional paths are required. This is shown in the following equation:

\[
\hat{A}bsW_r = AbsW_r \cup Conflict_r(Prop^q)
\]  \hspace{1cm} (4.6)

Notice that the refined abstract window, \(\hat{A}bsW_r\), is still an over-approximation of the concrete window because it only contains clauses that are either a subset, or implied by, the concrete window it models.

With the refined abstract window \(\hat{A}bsW_r\), we can similarly refine all previous abstract windows. This can be accomplished by re-creating Equation 4.5 with \(j < r\) and the current refined abstract window \((\hat{A}bsW_r)\). Since \(A\) is known to be invalid, each one of these instances will be \text{UNSAT} because they are just an extension of the assignment \(A\). After the abstract windows have been updated, the refined formula defined by updating Equation 4.4 can be solved for all solutions again. Similarly, these solutions can either be confirmed or used to refine the abstract time-windows until the refined instance, \(Abs\hat{D}ebug^q_p\), is unsatisfiable.

Once all solutions are found, \(Abs\hat{D}ebug^q_p\) is \text{UNSAT}, indicating that debugging has been completed on this time-window. The next theorem confirms the completeness of our approach; that is, the set of solutions found in the refined abstract model equals to the one found for the concrete model.

**Theorem 2** Let \(sols_{abs}\) be the set of confirmed debugging solutions returned by iteratively debugging and refining \(Abs\hat{D}ebug^q_p\) and let \(sols_{time}\) be the set of debugging solutions returned by \(Time\hat{D}ebug^q_p\). If the final refined abstract instance \(Abs\hat{D}ebug^q_p\) is \text{UNSAT} by blocking all solutions in \(sols_{abs}\), then \(sols_{abs} = sols_{time}\).

**Proof:** From Lemma 3, we know that \(sols_{abs} \supseteq sols_{time}\). From Lemma 4, we also know that any set of debugging solutions \(E\) found and verified by \(Abs\hat{D}ebug^q_p\) is also valid for \(Time\hat{D}ebug^q_p\) i.e., \(sols_{abs} \subseteq sols_{time}\). Therefore, \(sols_{abs}\) is both a superset and subset of \(sols_{time}\), so \(sols_{abs} = sols_{time}\).  

**Example 21** Building upon the abstract problem from Example 20, when the instance in Figure 4.4 is given to the SAT-solver, this instance returns one solution where \(e_a = 1\) since \(e_b\) was already blocked from the previous time-window. The associated cube with this satisfying
Assignment is cube\(^1\) = \(s_0^1 \land s_1^1 \land s_2^1\). This cube is verified by generating a new instance using Equation 4.5 with \(r = q = 1\), it is shown in Figure 4.5(a) and one can show that it is UNSAT. The clauses from the conflict graph are extracted and the abstract window AbsW\(_1^1\) is refined.

Figure 4.5(b) visualizes the refined abstract debugging problem Abs\(\hat{\text{Debug}}_{0^1}(N = 1)\). This instance again finds a solution with \(e_a = 1\) and a new cube, \(\hat{\text{cube}}^1 = s_0^1 \land s_1^1 \land s_2^1\). This cube is verified using Equation 4.5 which returns SAT. Thus \(e_a = 1\) is confirmed as a solution matching the results from Example 18. Once this solution is blocked, the refined instance returns UNSAT indicating that this time-window has completed analysis.
4.3.3 Overall Algorithm

Algorithm 4.2 presents pseudo-code for the path directed abstraction and refinement algorithm for time-windows from \( p \) to \( q \). The main loop from lines 3 to 13 iterates through each time-window analyzing them separately. Line 4 constructs the initial abstract instance using Equation 4.4 with any previously generated abstract windows (\( AbsW_i \)). In the first iteration, this equation simplifies to Equation 2.7 where the very last time-window is analyzed. The inner while loop (lines 5 to 11) finds all satisfying assignments and confirms each one by passing the result to the Verify procedure (Algorithm 4.3). If the assignment is not confirmed, then the procedure refines the relevant abstract windows. Once all the current assignments have been verified, the refined abstract problem is reconstructed (line 10), blocking any solutions that were confirmed from being found again. When the algorithm has finished analyzing the current time-window, it exits the while loop and generates the initial abstraction for the current window on line 12 for use in the next time-window.

The pseudo-code for the Verify procedure is shown in Algorithm 4.3. The procedure begins by extracting the state cube for window \( p + 1 \) from the assignment \( A \) on line 2. The outer for loop (lines 3 to 13) propagates the cube forward to subsequent windows using Equation 4.5 (line 4). If any of the subsequent time-windows are UNSAT (line 5), then the abstract windows are refined by iterating backwards through the windows (lines 6 to 9). Each backward iteration reconstructs Equation 4.5 with the refined abstract windows on line 7. The refinement step (line 8) extracts clauses from each of these UNSAT instances to update the abstract window. The procedure either returns the confirmed solution (line 14) or will return an empty solution otherwise (line 10).

4.4 Improved Refinement

A key step in Algorithm 4.2 is the Verify procedure which propagates state cube assignments forward to later time-windows. If any of the later windows are UNSAT, then the conflict graph can be used to extract clauses to refine the abstract windows. This section presents several methods to improve the basic refinement strategy to either reduce the overall number of iterations in Algorithm 4.2 or improve its performance.
Algorithm 4.2 Path Directed Abstraction and Refinement

1: procedure PathDebug
2: \hspace{1em} sols \leftarrow \emptyset
3: \hspace{1em} for \ p \in q, q - 1, \ldots, 1, 0 \ do
4: \hspace{2em} inst \leftarrow AbsDebug^q_p \land \text{BLOCK}(sols)
5: \hspace{2em} \text{while} \ inst \text{ is SAT do}
6: \hspace{3em} Assignments \leftarrow \text{SolveAll}(inst)
7: \hspace{3em} \text{for} \ A \in \text{Assignments} \ do
8: \hspace{4em} sols \leftarrow sols \cup \text{VERIFY}(A, p, q)
9: \hspace{3em} \text{end for}
10: \hspace{2em} inst \leftarrow AbsDebug^q_p \land \text{BLOCK}(sols)
11: \hspace{2em} \text{end while}
12: \hspace{1em} AbsW_p \leftarrow \text{Conflict}_p(inst)
13: \hspace{1em} \text{end for}
14: \hspace{1em} return sols
15: \hspace{1em} \text{end procedure}

4.4.1 Finding Additional Conflicts using Necessary Assignments

As described previously, given an UNSAT instance of Equation 4.5, one can extract conflicts to refine the current abstract window. However, since only one conflict is extracted per instance of Equation 4.5, this may require many calls to the VERIFY procedure, decreasing the overall performance of the algorithm. To mitigate this problem, one can extract multiple conflicts out of an UNSAT instance of Equation 4.5.

One method to accomplish this task is to remove literals in the propagated assignment (cube^i for \(i = r \cdot w\) in Equation 4.5) that were involved in the conflict and send the instance to the SAT solver again. If this less constrained problem is still UNSAT, then an additional conflict can be extracted to strengthen the refined abstraction. However, one drawback of this method is that the two conflicts will not have any overlapping literals in cube^i, i.e., the cube literals involved in the two conflicts are disjoint. This restricts the number of conflicts (and thus paths)
Algorithm 4.3 Solution Verification

1: procedure Verify(\(\mathcal{A}, p, q\))

2: \(\text{cube}^{(p+1)-w} \leftarrow \text{ExtractCube}(\mathcal{A}, p + 1)\)

3: for \(i \in p + 1, p + 2, \ldots, q - 1, q\) do

4: \(\text{inst} \leftarrow \text{Prop}_i^q\)

5: if \(\text{inst}\) is UNSAT then

6: for \(j \in i, i - 1, \ldots, p + 2, p + 1\) do

7: \(\text{inst} \leftarrow \text{Prop}_j^q\)

8: \(\text{AbsW}_j \leftarrow \text{AbsW}_j \cup \text{Conflict}_j(\text{inst})\)

9: end for

10: return \(\emptyset\)

11: end if

12: \(\text{cube}^{(i+1)-w} \leftarrow \text{ExtractCube}(\text{inst}, i + 1)\)

13: end for

14: return ExtractSuspect(\(\mathcal{A}\))

15: end procedure

that can be found since in many cases additional conflicts only occur when the literals overlap.

Alternatively, a more costly method would be to try all possible subsets of \(\text{cube}^i\) and see if any one of these results in an UNSAT instance. However, this method can be inefficient especially when the number of literals involved in the original conflict grows. A more efficient method for finding these types of conflicts is possible by determining additional information with regards to the literals of \(\text{cube}^i\).

Consider the ExtractCube procedure on line 2 of Algorithm 4.3. For each assignment \(\mathcal{A}\), we wish to store additional information about the extracted cube to distinguish which literals are due to the effect of the active error model \((e_i)\) from the assignment, and which are necessary regardless of the assignment to the error model’s free variables. When finding additional conflicts, one can simply remove only those cube literals that are not within this necessary assignment. This allows for additional overlapping conflicts from \(\text{cube}^i\) to be efficiently generated without the need to try all possible subsets of the cube literals.
Although it is possible to compute these necessary assignments to $cube^i$ precisely using multiple SAT calls \[96\], a more efficient approximate method is used. Consider the following equation of the abstract problem where the active suspect variable $e_i$ is constrained for a given assignment:

$$AbsDebug_j^q \land e_i$$

(4.7)

This instance has multiple SAT assignments due to the free variables from the error models disconnecting the active suspect’s fan-in from its fan-out. One important observation is that these free variables do not necessarily affect every state variable in the extracted cube ($cube^i$). Thus, an efficient method to determine which assignments are necessary is to run the solver’s Boolean constraint propagation (BCP) engine on the above instance. This will determine a large set of directly implied assignments to the extracted cube state variables ($cube^i$), thus producing a subset of the state cube that is necessary for every assignment involving $e_i$.

This same process can be repeated for Equation 4.5 as well by using only the necessary literals of the initial state cube:

$$Prop_j^q = Fixed(cube^{r-w}) \land W_r \land \bigwedge_{i=r+1}^q AbsW_i$$

(4.8)

Here, $Fixed$ extracts only the literals that were marked as necessary. Similarly, the solver’s BCP engine can be run on this instance to determine which literals in the resulting SAT assignment are necessary, and which ones are due to the effect of $e_i$. This will allow one to calculate the necessary variables for the next state cube $cube^{(r+1)-w}$.

Using this information for each UNSAT instance of Equation 4.5 additional overlapping conflicts can be extracted by removing non-necessary literals in $cube^i$ that were involved in the conflict, and re-running the SAT engine. This process can be repeated until either all non-necessary literals have been removed, or the instance becomes SAT.

### 4.4.2 Finding Additional Conflicts using Multiple State Cubes

As mentioned in the previous sub-section, each state cube ($cube^i$) consists of both literals that are affected by the error models, and those that are not affected (i.e., necessary literals). Since
the error model allows its corresponding circuit lines to be free, multiple assignments are possible to the subset of cube literals that are affected by the error models. In fact, given a cube\(^i\), the corresponding suspect variable may in fact be a solution but due to the approximate nature of the abstraction, a non-valid assignment to the free variables (with respect to the concrete problem) was chosen. This results in the subsequent propagation instances of Prop\(_j^q\) (i.e., Equation 4.5) to be UNSAT allowing refinement to take place. Eventually, the abstraction will be refined enough to exclude these spurious non-valid assignments; however, this may require many iterations. To overcome this issue, we can speculatively generate multiple additional state cubes from an original cube (cube\(^i\)) in an effort to find additional conflicts.

For a given cube\(^i\), an additional cube can be speculatively generated by inverting a non-necessary literal of the cube. The intuition here is that a single bit flip from a non-necessary literal has a high likelihood of being a valid solution to the abstract problem AbsDebug\(_p^q\) (and thus a valid cube). However, it is possible that this assignment is not a solution to the abstract problem. In this case, it is not relevant so long as the speculatively generated cube results in a conflict to allow additional paths to be refined. These additional paths may not necessarily be used directly for the current suspect variable \(e_i\), but may be useful in later time-windows or for other suspect variables. Since the conflicts are still with respect to Equation 4.5, any refinement will maintain the over-approximate nature of the abstraction. This results in a net benefit because these additional cubes are efficient to calculate and accelerate refinement.

The additional cubes can be applied to Equation 4.5 on line 7 of Algorithm 4.3. If the resulting instances are UNSAT, additional clauses can be extracted from the conflict-graph accelerating refinement in an efficient manner.

4.4.3 Improving the Initial Abstraction

The initial abstraction for each time-window (as calculated on line 12 of Algorithm 4.2) is derived from a single conflict graph. This is typically a very weak abstraction because it contains only a small subset of paths involved in the original failure. An efficient means to generate a stronger initial abstraction would greatly increase performance by reducing the number of refinement iterations needed to analyze later time-windows.
Observe that each abstract time-window has the same unrolled circuit, differing only in the error trace values that are applied. This leads to the idea that many of the paths that are needed for an error to propagate are similar between different time-windows. More precisely, we can use the previously refined abstract time-windows ($\hat{AbsW}_{p+1}$) to strengthen the initial abstraction for the current time-window ($AbsW_p$).

For $AbsW_p$, many of the clauses from the previous refined window, $\hat{AbsW}_{p+1}$, will be valid. We can use these clauses to strengthen $AbsW_p$ by shifting the literals in the respective clauses to the corresponding circuit lines in the current window. However, not all of the clauses will be valid due to differences in the error trace values. Therefore, each clause from $\hat{AbsW}_{p+1}$ must be checked in order to ensure its validity. This can be accomplished efficiently in the following way.

For each clause in $\hat{AbsW}_{p+1}$, remap its variables so they correspond to the same circuit lines in the current window ($AbsW_p$). Next, negate the clause and add the corresponding literals to the abstract debug problem as shown in the following equation:

$$AbsDebug_q^p \land \overline{\text{clause}_i}$$  \hspace{1cm} (4.9)

If this instance is UNSAT, then the clause is implied by the abstract problem and can be added to the current abstract window ($AbsW_p$). Since this needs to be performed for many clauses, an efficient approximate method to accomplish this is to use the SAT solver’s BCP engine to propagate the literals in $\overline{\text{clause}_i}$. If the result is UNSAT, then we can add the clause to the current abstract window. Otherwise, do not add it. This will not necessarily determine if the clause is implied by the abstract problem in all cases, but rather provides an efficient means to gather a large subset of them compared to explicit calls to the SAT-solver.

**Example 22** Consider the abstract instance $AbsDebug_q^p$ involving these clauses:

$$AbsDebug_q^p = (a + b)(\bar{b} + c)(\bar{a} + \bar{d})$$

Suppose, we wish to determine if clause $(c + d)$ and $(c + \bar{d})$ can be implied by this instance. We
can simply perform BCP on the following two instances:

\[(a + b)(\overline{b} + c)(\overline{a} + d)(\overline{c})(\overline{d})\]
\[(a + b)(\overline{b} + c)(\overline{a} + d)(\overline{c})(d)\]

The former case yields no conclusive results, so it is skipped. The latter case is UNSAT, allowing us to deduce that \((c + d)\) is implied by \(\text{AbsDebug}\_q^p\).

### 4.4.4 Leveraging the SAT-solver for Efficient Implementation

Beyond improvements to refinement, an essential aspect in realizing the proposed algorithm is making extensive use of the SAT-solver. Two key implementation notes are discussed in this sub-section. The first improvement involves extensive use of unit assumptions and incremental SAT capabilities of modern solvers [19, 53]. This capability of modern solvers is used wherever possible to decrease the number of newly created instances, greatly reducing the computational friction between different SAT instances. One example of where this provided significant benefit is a modification of Algorithm 4.2. During the \text{Verify} procedure, instead of verifying one assignment at a time, multiple assignments can be verified using one instance of \(\text{Prop}\_q^i\). This is accomplished by setting each cube, \(\text{cube}^i\cdot w\), as unit assumptions and re-solving the instance using incremental SAT. This greatly reduces the overhead of re-solving the same instance (with the exception of the state cube) multiple times.

The second improvement involves clause subsumption [19]. During refinement, many new clauses are added to the abstraction. Some are redundant due to overlapping paths. To ensure that the abstraction does not contain this unnecessary bloat, subsumption is used to reduce duplicate information being stored. This step can be performed each time an abstract window is refined, maintaining an efficient abstraction.

### 4.5 Flexible Path Directed Debug

Algorithm 4.2 provides a complete method to find the exact set of suspects as the explicit formulation of Equation 4.2 while dramatically reducing the problem size. However in certain worse-case scenarios, the iterative nature of calling the \text{VERIFY} procedure will result in long
Algorithm 4.4 Flexible Path Directed Abstraction and Refinement

1: procedure FlexPathDebug
2:     sols ← ∅, skips ← ∅
3:     for p ∈ q, q − 1, . . . , 1, 0 do
4:         inst ← AbsDebug_q ∧ Block(sols)
5:         while inst is SAT do
6:             Assignments ← SolveAll(inst)
7:             for A ∈ Assignments do
8:                 if count_sols(A) + < skip_limit then
9:                     sols ← sols ∪ Verify(A, p, q)
10:                else
11:                    skips ← skips ∪ sol(−A)
12:                end if
13:             end for
14:         inst ← AbsDebug_q ∧ Block(sols ∪ skips)
15:     end while
16:     AbsW_p ← Conflict_p(inst)
17: end for
18: return sols, skips
19: end procedure

run-times. In this section, we introduce a flexible algorithm that allows a trade-off between performance and quality of results to mitigate these problematic cases.

The worse-case scenario in Algorithm 4.2 occurs when an excessive number of iterations of the while loop on line 5 happen. This can happen when a suspect requires an excessive number of paths to prove that it is valid. The algorithm will continually attempt to refine the abstract problem and will eventually converge to prove or disprove the solution’s validity but may require a long run-time.

To overcome this issue, consider the case where a suspect is verified on an abstract model. The fact that it is verified on the abstract model does give some confidence that it is valid,
especially if it is on a partially refined model. This translates to the suspect being valid among a subset of paths that were present in the abstraction, but not necessarily among all paths in the concrete model.

Using this idea, if a suspect has been propagated forward multiple times without ruling it out, then it can be recorded as partially verified, and skipped for future iterations of the algorithm. In this manner, we remove the worse-case scenario by limiting how many times a given assignment for a suspect can be propagated forward. Practically, this results in a small number of suspects being skipped while providing a large benefits in run-time.

Algorithm 4.4 shows pseudo-code for the modified algorithm named Flexible PathDebug. This algorithm defines two new variables, the \textit{skip\_limit} parameter and \textit{skips} set. The \textit{skip\_limit} parameter defines the trade-off between quality of results and performance. In the case of \textit{skip\_limit} = \infty, the behavior mimics Algorithm 4.2. In the case of \textit{skip\_limit} = 0, no verification is done on the solutions resulting in a similar algorithm to \cite{82} where no refinement is present. For values in between these two limits, the user can define a refinement strategy that allows a trade-off between complete results and a loose abstraction. The \textit{skip} set contains all suspects that were skipped due to the \textit{skip\_limit}. This distinguishes them from the suspects that have been fully verified giving additional information to the user.

In Algorithm 4.4, the main loop along with the \textbf{while} loop retain the same functionality of analyzing time-windows and verifying satisfying assignments. However, the algorithm differs on line 8 where it checks if the assignment’s suspect has reached the \textit{skip\_limit}. If it is within the limit, the algorithm runs the \textsc{Verify} procedure as before (line 9). Otherwise, the suspect associated with this solution is added to the \textit{skips} set (line 11). The \textit{skips} set is then blocked on line 14 so that it does not show up again as an assignment in the \textbf{while} loop. Finally, the algorithm will return both the \textit{sols} and \textit{skips} set to the user in order to distinguish between solutions that have been verified and ones that have not.

### 4.6 Experiments

This section presents the experimental results of the proposed algorithms. The same general experimental setup as Chapter 3 is used for this chapter with only several parameters changed.
Chapter 4. Path Directed Abstraction and Refinement

In this chapter, all experiments are run on a single core of an Intel Core i5 3.1 Ghz quad-core workstation with 8 GB of RAM and a timeout of 7200 seconds. All experiments use MINISAT [53] as the SAT solver with a commercial Verilog frontend [139] to allow for diagnosis of RTL designs.

The effectiveness of the techniques are shown on industrial Verilog RTL designs from OpenCores [111] as well as two commercial designs (fxu, comm) provided by our industrial partners. Table 4.1 presents relevant design statistics for each of the RTL designs. The three columns show the design name, number of gates, and the total number of potential error locations where error models are applied. The designs in this table may be used in experiments for other chapters, but may differ in their design characteristics. This difference can attributed to the version of the design, the version of the Verilog frontend, and the use of a cone-of-influence reduction described below.

Each debugging instance is created by randomly selecting a line in the RTL and inserting a typical industrial RTL error (wrong state transition, incorrect operator, erroneous instantiation etc.). It is important to emphasize that each RTL error typically corresponds to multiple gate-level errors locations, effectively allowing for diagnosis of multiple errors. Next, the buggy RTL is simulated with its accompanying testbench with a commercial simulator [101] to observe the failure and record its error trace. The next step runs each instance through a cone of influence reduction [35] to remove logic that is not involved in the observed failure. This is performed for all instances in all experiments and may result in different effective problem sizes for different instances of the same design. Finally, each instance is run through the respective debugging algorithm with $N = 1$ to compare the performance and quality of results.

4.6.1 Refinement Improvements

The first set of experiments shows the effect of the refinement improvements from Section 4.4. The results are shown in Table 4.2. The basic algorithm from Section 4.3 is compared to the same algorithm but with the refinement improvements from Section 4.4.1 to 4.4.3. To ensure comparable results, instances are selected such that they are able to run to completion without causing a time-out or mem-out using 10 time-windows of size $w = 10$ (i.e., an error trace of
Table 4.1: Design Characteristics

<table>
<thead>
<tr>
<th>design</th>
<th># gates (k)</th>
<th># suspect locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac97ctrl</td>
<td>22.9</td>
<td>1380</td>
</tr>
<tr>
<td>comm</td>
<td>164.4</td>
<td>20155</td>
</tr>
<tr>
<td>div64bits</td>
<td>59.7</td>
<td>197</td>
</tr>
<tr>
<td>fdct</td>
<td>239.5</td>
<td>4662</td>
</tr>
<tr>
<td>fpu</td>
<td>72.5</td>
<td>1982</td>
</tr>
<tr>
<td>fxu</td>
<td>447.3</td>
<td>33087</td>
</tr>
<tr>
<td>mips</td>
<td>50.3</td>
<td>2636</td>
</tr>
<tr>
<td>rsdecoder</td>
<td>14.5</td>
<td>2040</td>
</tr>
<tr>
<td>usbfunct</td>
<td>35.8</td>
<td>4061</td>
</tr>
<tr>
<td>vga</td>
<td>48.8</td>
<td>1731</td>
</tr>
</tbody>
</table>

100 time-frames).

The first column in Table 4.2 shows the instance name. The next six columns show the run-time, peak memory usage, and number of calls to Verify procedure in Algorithm 4.3 for both the basic algorithm and the one with the refinement improvements.

The benefits of the refinement improvements manifest themselves in both the run-time and number of calls to Verify with an average decrease of 33.4% and 66.0% respectively. The average peak memory increased slightly by 1.5%, which is an acceptable trade-off for the increased performance. The reduction in run-time is primarily caused by the reduced number of calls to Verify showing the efficacy of the improvements to quickly refine the abstraction.

From Table 4.2 the refinement improvements do incur some overhead as can be seen in ac97ctrl.1, where the number of calls to Verify are identical, but the improvements take longer to run. This can be explained by the fact that ac97ctrl.1 case does not require much refinement, and the improvements have little effect in these cases. However for most of the other instances, the extra work required for the refinement improvements show a large reduction in run-time improvement. For usbfunct.2, the refinement improvements from Section 4.4.3 led to no calls to the Verify procedure. This was caused by all valid solutions being found in the first analyzed time-window (latest in time), as well as the improvements creating a strong initial abstraction for each time-window which filtered out any potential spurious solutions. However, without the refinement improvements, spurious solutions are generated that must then be excluded through further refinement by calls to the Verify procedure.
Table 4.2: Experiments showing the effect of the proposed optimizations from Section 4.4.

<table>
<thead>
<tr>
<th>instance</th>
<th>time (s)</th>
<th>peak memory (MB)</th>
<th>Number of calls to Verify</th>
<th>time (s)</th>
<th>peak memory (MB)</th>
<th>Number of calls to Verify</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac97ctrl_1</td>
<td>47</td>
<td>703</td>
<td>14</td>
<td>54</td>
<td>722</td>
<td>14</td>
</tr>
<tr>
<td>ac97ctrl_2</td>
<td>132</td>
<td>716</td>
<td>115</td>
<td>81</td>
<td>775</td>
<td>29</td>
</tr>
<tr>
<td>div64bits_1</td>
<td>392</td>
<td>1366</td>
<td>110</td>
<td>275</td>
<td>1368</td>
<td>55</td>
</tr>
<tr>
<td>div64bits_2</td>
<td>206</td>
<td>1352</td>
<td>213</td>
<td>72</td>
<td>1270</td>
<td>6</td>
</tr>
<tr>
<td>fpu_1</td>
<td>108</td>
<td>1450</td>
<td>25</td>
<td>59</td>
<td>1475</td>
<td>3</td>
</tr>
<tr>
<td>rsdecoder_2</td>
<td>1354</td>
<td>729</td>
<td>1468</td>
<td>1582</td>
<td>861</td>
<td>552</td>
</tr>
<tr>
<td>usbfunct_1</td>
<td>106</td>
<td>1122</td>
<td>29</td>
<td>83</td>
<td>1102</td>
<td>20</td>
</tr>
<tr>
<td>usbfunct_2</td>
<td>88</td>
<td>1051</td>
<td>36</td>
<td>45</td>
<td>1040</td>
<td>0</td>
</tr>
<tr>
<td>vga_2</td>
<td>3544</td>
<td>1155</td>
<td>3733</td>
<td>657</td>
<td>1060</td>
<td>352</td>
</tr>
</tbody>
</table>

Table 4.3: Experiments showing the effect of window size on performance.

<table>
<thead>
<tr>
<th>instance</th>
<th>w = 5</th>
<th>w = 10</th>
<th>w = 25</th>
<th>w = 50</th>
<th>w = 100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time (s)</td>
<td>peak mem (MB)</td>
<td>time (s)</td>
<td>peak mem (MB)</td>
<td>time (s)</td>
</tr>
<tr>
<td>ac97ctrl_1</td>
<td>80</td>
<td>692</td>
<td>54</td>
<td>722</td>
<td>39</td>
</tr>
<tr>
<td>ac97ctrl_2</td>
<td>104</td>
<td>699</td>
<td>81</td>
<td>775</td>
<td>51</td>
</tr>
<tr>
<td>div64bits_1</td>
<td>531</td>
<td>1250</td>
<td>275</td>
<td>1368</td>
<td>161</td>
</tr>
<tr>
<td>div64bits_2</td>
<td>244</td>
<td>1225</td>
<td>72</td>
<td>1270</td>
<td>75</td>
</tr>
<tr>
<td>fpu_1</td>
<td>82</td>
<td>1279</td>
<td>59</td>
<td>1475</td>
<td>58</td>
</tr>
<tr>
<td>rsdecoder_2</td>
<td>1875</td>
<td>873</td>
<td>1582</td>
<td>861</td>
<td>1494</td>
</tr>
<tr>
<td>usbfunct_1</td>
<td>124</td>
<td>961</td>
<td>83</td>
<td>1102</td>
<td>58</td>
</tr>
<tr>
<td>usbfunct_2</td>
<td>72</td>
<td>954</td>
<td>45</td>
<td>1040</td>
<td>36</td>
</tr>
<tr>
<td>vga_2</td>
<td>2709</td>
<td>1136</td>
<td>657</td>
<td>1060</td>
<td>671</td>
</tr>
</tbody>
</table>

4.6.2 Window Size

The next set of experiments profiles the effects when varying the width of the time-windows ($w$). The same nine instances from Section 4.6.1 are used to ensure that all instances are able to run to completion without causing a time-out or memory-out condition for the given window sizes. The trace size is set to 100, while varying the width of each window for $w = 5, 10, 25, 50, 100$. In the case of $w = 100$, the problem does not use a time-windowing technique and models the entire trace directly.

Table 4.3 shows the run-time and peak memory results of the experiments. The first column
Chapter 4. Path Directed Abstraction and Refinement

shows the instance name, followed by the run-time in seconds and peak memory for each of the different values of $w$. Overall, using a time-windowing technique dramatically reduces the peak memory usage of the algorithm, resulting in an average decrease of 62.0%, 60.1%, 52.0%, and 36.4% in peak memory for $w = 5, 10, 25, 50$, respectively, compared to $w = 100$ (i.e., no time-windowing). Using a geometric mean, the respective averages are 0.36x, 0.39x, 0.47x and 0.63x peak memory compared to $w = 100$. This decrease is directly related to the reduction in problem size due to using time-windowing. Figure 4.6 shows this relative decrease in peak memory for several instances.

As the window width $w$ shrinks, the peak memory does not approach $\frac{1}{w}$ of the peak memory of directly modeling the entire trace. This is caused by two reasons. First, the data structures used to hold the RTL constructs, error trace, and internal netlist are always present in memory resulting in a lower bound regardless of window size. Secondly, the abstraction is still present in memory. Regardless of how many time-frames are explicitly modeled, the abstraction must still contain enough clauses in order to verify each solution. This places an additional constraint on the lower bound of the problem. Despite this bound, the time-windowing technique still produces dramatic decreases in peak memory.

When comparing the run-time, the results show an overall increase when compared to using no time-windowing ($w = 100$). For $w = 5, 10, 25, 50$, time-windowing increases run-time on average by 216.6%, 46.0%, 25.1%, and 183.54%, respectively. Using a geometric mean, the respective averages are 0.54x, 0.97x, 1.2x, and 0.92x speedup compared to $w = 100$. In some cases time-windowing produced faster run-times such as div64bits_1, and in other cases slower run-times such as vga_2. The former case occurs primarily when the abstract problem is small enough allowing the solver to run faster to outweigh the overhead of multiple SAT instances. In the latter case, multiple SAT instances increase the run-time, especially in cases where many calls to VERIFY are needed. Here explicit modeling of the trace ($w = 100$) produces significantly better results in terms of run-time due to the solver having access to the entire problem in memory. However, it should be noted that the primary benefit of time-windowing techniques are targeted for large industrial instances where it is not possible to model the entire trace explicitly. In these cases, time-windowing makes analysis possible while explicit modeling
is unable to handle these problems.

The run-time performance of an instance across different window widths varies significantly as in the case of vga\textunderscore 2. The run-time is dependent primarily on the number of SAT instances that are created and the size of each one of those instances. For smaller window widths, the size of each instance is small and runs quickly, but many small SAT instances are created (primarily in the VERIFY procedure). Conversely for longer window widths, the size of each instance is large and takes longer to run, but the number of SAT instances is significantly less. In addition to these two effects, the conflict driven refinement procedure is dependent on the solution found in the abstract problem. This adds additional variability, where in a worst cast, significantly more calls to the VERIFY procedure may be needed. However, as shown in the next section, using the flexible algorithm from Section 4.5 (Algorithm 4.4), this variability can be reduced.

### 4.6.3 Flexible Path Directed Debug

This set of experiments investigates the FLEXPATHDEBUG algorithm from Algorithm 4.4 with respect to the skip\_limit parameter. Instances are chosen that are able to run to completion for all values of skip\_limit without a time-out or memory-out condition. The trace is set to analyze 100 time-frames with $w = 10$. Four different sets of experiments are conducted for skip\_limit = 0, 5, 10, $\infty$. Note that skip\_limit = $\infty$ corresponds to the basic PATHDEBUG
algorithm from Algorithm 4.2, while \( \text{skip\_limit} = 0 \) is comparable to the approximate time-windowing technique from [82] where no refinement is present.

Table 4.4 shows the results of the experiments. The first column shows the instance name, followed by four sets of columns corresponding to the run-time, peak memory, number of verified and un-verified solutions (\( i.e., \text{sols} \) and \( \text{skips} \) from Algorithm 4.4) for the four different values of \( \text{skip\_limit} \).

When looking at the \( \text{skip} \) column, small values of \( \text{skip\_limit} \) have a large effect on the number of un-verified solutions. On average, the number of skipped solutions are 114.7%, 1.6% and 1.5% times the size of \( \text{skip\_limit} = \infty \) (\( i.e., \) the exact set of solutions) for \( \text{skip\_limit} = 0, 5, 10 \), respectively.

This shows that refinement can greatly reduce the number of spurious solutions compared to past approximate time-windowing techniques such as [82] where no refinement is present. On the other hand, even for small values of \( \text{skip\_limit} \) the number of skipped solutions is greatly reduced.

In terms of run-time, the average run-time reduction for \( \text{skip\_limit} = 0, 5, 10 \) relative to \( \text{skip\_limit} = \infty \) is 52.4%, 9.72% and 7.5%, respectively. Using a geometric mean, the respective averages are 2.97x, 1.13x, and 1.09x speedup compared to \( \text{skip\_limit} = \infty \). As expected \( \text{skip\_limit} = 0 \) is able to show the greatest run-time reduction at the cost of greatly decreased quality of results. For other values of \( \text{skip\_limit} \), the refinement can dramatically reduce the run-time as in the case of \text{comm.1} \ and \text{vga.2}, where a small subset of suspects causes the algorithm to spend significantly more time in the \text{VERIFY} procedure. This worst-case scenario is exactly the case \text{PATHDEBUG} aims to address.

When looking at peak-memory, the average reduction is 5.4%, 0.6%, and 0.5% for \( \text{skip\_limit} = 0, 5, 10 \) respectively. Using a geometric mean, the respective averages are 0.95x, 0.99x, 0.99x peak memory compared to \( \text{skip\_limit} = \infty \). When no refinement is present (\( \text{skip\_limit} = 0 \)), a slight decrease in peak memory is shown due to the fact that the abstraction is smaller compared to when refinement is present.
4.6.4 Trace Length

The final set of experiments investigates the maximum length of error trace each technique can analyze until either a time-out, memory-out or entire trace is analyzed. A time-out occurs whenever the process uses more than 7200 seconds. When this situation occurs, a given time-window is likely to be in the middle of analysis where several solutions have been found but not necessarily all solutions. In this case, any partial solutions that have been found in this time-window are included in the results because they are still useful even though all solutions have not been found. A memory-out occurs whenever the process uses more than 8 GB of RAM. This typically occurs when analysis moves onto a new time-window which increases the size of the abstract debug problem in memory.

The experiments compare the WindowExpansion time-windowing technique from [82] (a complete debugging algorithm), the basic PathDebug algorithm from Section 4.3 and the flexible algorithm FlexPathDebug from Section 4.5 with \( \text{skip\_limit} = 5 \). All instances derived from designs in Table 4.1 are used in this set of experiments. The window size is set to \( w = 10 \).

Table 4.5 shows the results of the experiments. The first column shows the instance name, while the next eight columns show the run-time, peak memory, number of solutions, and number of cycles analyzed for the WindowExpansion and PathDebug algorithms, respectively. The last five columns show the run-time, peak memory, number of verified and un-verified solutions (i.e., \( \text{sols} \) and \( \text{skips} \) from Algorithm 4.4), and number of analyzed cycles.

When comparing the number of cycles analyzed, PathDebug and FlexPathDebug are able to analyze 35.5% and 64.6% more cycles on average than WindowExpansion. Using a geometric mean, the averages are 1.10x and 1.43x increase, respectively. This shows one of the main benefits of the proposed technique: the ability to analyze debugging instances that were previously too large to analyze. Moreover, the increased trace length analyzed for PathDebug and FlexPathDebug is achieved with significantly less memory. This can be seen by WindowExpansion experiments causing 12 memory-out conditions compared to 2 for the path-based algorithms.

Figure 4.7 shows the cycles analyzed for several different instances. In most cases, such as
fdct.1 and comm.1, the FlexPathDebug is able to analyze many more cycles compared to WindowExpansion. In contrast, in a few cases such as mips.1, WindowExpansion is able to analyze more cycles. This is due to the computation required by the Verify procedure, where the SAT instances created are more difficult to solve. However, the FlexPathDebug algorithm can be used with a different value of skip_limit parameter to overcome some of these hard-to-solve cases.

For the FlexPathDebug algorithm, the absolute number of un-verified solutions is relatively small allowing for a good trade-off between performance and quality of results. Moreover, the additional cycles analyzed allowed FlexPathDebug to find on average 11.6% more solutions (both verified and un-verified) compared to WindowExpansion (1.10x using a geometric mean). This shows that the additional cycles analyzed do indeed contain some new solutions, which could potentially be needed to find the underlying bug, a desirable side-effect for the proposed methodology.

### 4.7 Summary

This chapter presents a novel path directed abstraction and refinement framework that aims to manage excessive error trace lengths. It uses a time-windowing framework to iteratively
analyze a sliding window of the error trace. Non-modeled portions of the trace are approximated using a novel path directed abstraction that conceptually represents structural circuit paths. This representation provides a means for analyzing a sliding window of the error trace, as well providing a basis for efficient refinement. The result is an algorithm that can significantly reduce the memory requirements of debugging industrial instances, while mitigating the incomplete results of past techniques. Experimental results on industrial designs demonstrates that 64.6% longer error traces can be analyzed with the proposed approach while significantly reducing the memory usage.
Table 4.4: Experiments for FlexPathDebug (Algorithm 4.4) showing the effect of various skip limit values on run-time, memory, number of verified solutions and number of skipped solutions.

<table>
<thead>
<tr>
<th>instance</th>
<th>skip limit = 0</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time (s)</td>
<td>peak mem (MB)</td>
<td># sols</td>
<td># skip sols</td>
<td>time (s)</td>
<td>peak mem (MB)</td>
<td># sols</td>
<td># skip sols</td>
<td>time (s)</td>
<td>peak mem (MB)</td>
<td># sols</td>
<td># skip sols</td>
<td>time (s)</td>
</tr>
<tr>
<td>ac97ctrl_1</td>
<td>27</td>
<td>720</td>
<td>15</td>
<td>11</td>
<td>54</td>
<td>722</td>
<td>26</td>
<td>0</td>
<td>54</td>
<td>722</td>
<td>26</td>
<td>0</td>
<td>54</td>
</tr>
<tr>
<td>ac97ctrl_2</td>
<td>25</td>
<td>722</td>
<td>9</td>
<td>11</td>
<td>81</td>
<td>775</td>
<td>11</td>
<td>0</td>
<td>81</td>
<td>775</td>
<td>11</td>
<td>0</td>
<td>81</td>
</tr>
<tr>
<td>div64bits_1</td>
<td>146</td>
<td>1334</td>
<td>23</td>
<td>17</td>
<td>275</td>
<td>1368</td>
<td>36</td>
<td>0</td>
<td>275</td>
<td>1368</td>
<td>36</td>
<td>0</td>
<td>275</td>
</tr>
<tr>
<td>div64bits_2</td>
<td>63</td>
<td>1260</td>
<td>18</td>
<td>3</td>
<td>72</td>
<td>1270</td>
<td>21</td>
<td>0</td>
<td>72</td>
<td>1270</td>
<td>21</td>
<td>0</td>
<td>72</td>
</tr>
<tr>
<td>fpu_1</td>
<td>50</td>
<td>1444</td>
<td>29</td>
<td>3</td>
<td>59</td>
<td>1475</td>
<td>32</td>
<td>0</td>
<td>59</td>
<td>1475</td>
<td>32</td>
<td>0</td>
<td>59</td>
</tr>
<tr>
<td>rsdecoder_2</td>
<td>54</td>
<td>579</td>
<td>132</td>
<td>275</td>
<td>1582</td>
<td>861</td>
<td>381</td>
<td>0</td>
<td>1582</td>
<td>861</td>
<td>381</td>
<td>0</td>
<td>1582</td>
</tr>
<tr>
<td>comm_1</td>
<td>1139</td>
<td>5152</td>
<td>32</td>
<td>201</td>
<td>3442</td>
<td>5176</td>
<td>74</td>
<td>2</td>
<td>3986</td>
<td>5248</td>
<td>74</td>
<td>2</td>
<td>6169</td>
</tr>
<tr>
<td>comm_2</td>
<td>1062</td>
<td>5190</td>
<td>32</td>
<td>203</td>
<td>3449</td>
<td>5315</td>
<td>75</td>
<td>2</td>
<td>3981</td>
<td>5318</td>
<td>75</td>
<td>2</td>
<td>6168</td>
</tr>
<tr>
<td>usbfunct_1</td>
<td>52</td>
<td>1080</td>
<td>98</td>
<td>11</td>
<td>86</td>
<td>1099</td>
<td>105</td>
<td>1</td>
<td>83</td>
<td>1102</td>
<td>105</td>
<td>0</td>
<td>83</td>
</tr>
<tr>
<td>usbfunct_2</td>
<td>45</td>
<td>1040</td>
<td>10</td>
<td>0</td>
<td>45</td>
<td>1040</td>
<td>10</td>
<td>0</td>
<td>45</td>
<td>1040</td>
<td>10</td>
<td>0</td>
<td>45</td>
</tr>
<tr>
<td>vga_2</td>
<td>101</td>
<td>1033</td>
<td>28</td>
<td>195</td>
<td>514</td>
<td>1063</td>
<td>46</td>
<td>5</td>
<td>580</td>
<td>1053</td>
<td>46</td>
<td>5</td>
<td>657</td>
</tr>
</tbody>
</table>
Table 4.5: Experiments to determine the maximum trace length each algorithm can analyze. It benchmarks the WindowExpansion algorithm from [82], the basic PathDebug algorithm from Section 4.3, and the flexible algorithm FlexPathDebug from Section 4.5 with skip_limit = 5. TO and MO indicate time-out and memory-out conditions, respectively.

<table>
<thead>
<tr>
<th>instance</th>
<th>WindowExpansion [82]</th>
<th>PathDebug</th>
<th>Flexible PathDebug</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time (s)</td>
<td>peak mem (MB)</td>
<td># cycles analyzed</td>
</tr>
<tr>
<td>ac97ctrl_1</td>
<td>2812</td>
<td>MO 46 600</td>
<td>TO 3830 51 1280</td>
</tr>
<tr>
<td>ac97ctrl_2</td>
<td>4238</td>
<td>MO 122 540</td>
<td>TO 3955 123 1350</td>
</tr>
<tr>
<td>fxu_1</td>
<td>790</td>
<td>MO 174 10</td>
<td>925 MO 174 40</td>
</tr>
<tr>
<td>fxu_2</td>
<td>295</td>
<td>MO 37 10</td>
<td>354 MO 37 10</td>
</tr>
<tr>
<td>div64bits_1</td>
<td>419</td>
<td>MO 3971 100</td>
<td>TO 1551 36 100</td>
</tr>
<tr>
<td>div64bits_2</td>
<td>383</td>
<td>MO 3517 21</td>
<td>72 MO 1301 21</td>
</tr>
<tr>
<td>fdct_1</td>
<td>2491</td>
<td>MO 78 40</td>
<td>TO 4798 80</td>
</tr>
<tr>
<td>fdct_2</td>
<td>3476</td>
<td>MO 69 40</td>
<td>TO 4214 69</td>
</tr>
<tr>
<td>fpu_1</td>
<td>797</td>
<td>MO 35 210</td>
<td>340 MO 2419 230</td>
</tr>
<tr>
<td>fpu_2</td>
<td>40</td>
<td>1944 5</td>
<td>27 1000 5</td>
</tr>
<tr>
<td>mips_1</td>
<td>2375</td>
<td>7454 126</td>
<td>250 1298 103</td>
</tr>
<tr>
<td>mips_2</td>
<td>6104</td>
<td>MO 260 280</td>
<td>TO 1242 80</td>
</tr>
<tr>
<td>rsdecoder_1</td>
<td>TO 1849</td>
<td>81 60</td>
<td>TO 1216 75</td>
</tr>
<tr>
<td>rsdecoder_2</td>
<td>548</td>
<td>1093 384</td>
<td>100 2473 1051</td>
</tr>
<tr>
<td>comm_1</td>
<td>990</td>
<td>MO 72 60</td>
<td>TO 5515 74</td>
</tr>
<tr>
<td>comm_2</td>
<td>1113</td>
<td>MO 73 60</td>
<td>TO 5490 75</td>
</tr>
<tr>
<td>usbfunct_1</td>
<td>3788</td>
<td>MO 105 410</td>
<td>351 2016 105</td>
</tr>
<tr>
<td>usbfunct_2</td>
<td>4248</td>
<td>MO 106 410</td>
<td>2777 3186 106</td>
</tr>
<tr>
<td>vga_1</td>
<td>TO 4489</td>
<td>20 100</td>
<td>TO 1890 20</td>
</tr>
<tr>
<td>vga_2</td>
<td>TO 3790</td>
<td>46 240</td>
<td>TO 1422 46</td>
</tr>
</tbody>
</table>
Chapter 5

Debugging of Multiple Errors Using UNSAT Cores

5.1 Introduction

Design debugging has three key parameters that affect the complexity of the problem: design size, error trace length, and error cardinality (i.e., the number of simultaneous errors that cause a failure). The error cardinality is perhaps the most daunting because the solution space of the debug problem grows exponentially with the number of errors [137].

A previous algorithm [131] using multiple unsatisfiable (UNSAT) cores has shown to be effective in tackling this problem. In this situation, each core intuitively represents a tree of paths in the circuit that is responsible for causing the observed error. By analyzing the intersections of these cores, the search space can be dramatically reduced, improving the performance of the debugging engine. However, the main benefit of the technique for multiple design errors relies on the ability to find multiple UNSAT cores. This is a very difficult problem with no efficient solution in general [110]. In practice, only disjoint UNSAT cores can be found, which limits the technique’s ability to be more widely applicable.

In this chapter, we present a new algorithm to deal with debugging multiple design errors using unsatisfiable cores. It generalizes previous work by showing how to both generate and apply unsatisfiable cores for multiple design errors. The process begins by generating an UNSAT
core at error cardinality zero from an unsatisfiable SAT-based debugging instance. The core is used to constrain the search space at the next cardinality, which is then solved for all solutions. After these solutions are blocked, another UNSAT core is generated from this instance. This process is repeated at each subsequent cardinality. By using the cores generated at each cardinality, we can rule out locations that cannot be part of solutions at higher error cardinalities. This effectively reduces the search space for multiple design errors, which manages the exponential growth in solution space. In addition, since the solving process generates these cores as a by-product, there is little overhead needed to gain significant benefit in reducing the search space.

Experimental results on large hardware designs from OpenCores [111] show the efficacy of the proposed algorithm compared to previous work. For finding all equivalent errors up to error cardinality three, the core technique is able to reduce the total run-time on average by 22% with a negligible impact on peak memory compared to previous work.

The remaining sections are organized as follows. Section 5.2 describes background information. Section 5.3 describes the main contribution while Section 5.4 presents the experimental results. Finally, Section 5.5 concludes this work.

5.2 Preliminaries

5.2.1 Unsatisfiable Cores in Design Debugging

Recall from Section 2.4.2, an unsatisfiable (UNSAT) core is an unsatisfiable subset of clauses of an unsatisfiable Boolean formula written in CNF. Modern SAT solvers can produce UNSAT cores once they determine that an instance is unsatisfiable [146]. In the context of design debugging, an UNSAT core intuitively represents a tree of paths in the circuit from which the primary inputs and initial states propagate to the expected primary outputs and result in a conflict.

This idea has been used in several algorithms to generate information to aid in the process of SAT-based design debugging [82, 83, 131]. The work in [131] uses the above concept of UNSAT cores to restrict which areas of the circuit need to be examined. If an UNSAT core exists, at
least one of the suspect variables in the core must be active to break the conflict. For multiple UNSAT cores, at least one suspect variable in each core must be active to break the conflict.

Figure 5.1 shows this concept in more detail. Each UNSAT core defines a region of the potential suspect space corresponding to which suspect variables are present in the core. Any solution to the SAT-based debugging formulation must break each one of the UNSAT cores, therefore every solution must contain at least one suspect from each core. Using this idea, any \( N = 1 \) solution must exist in the intersection of both Core 1 and Core 2, and any \( N = 2 \) solution must contain suspects from the non-intersecting parts of both Core 1 and Core 2. In the general case, any solution must contain suspects such that all UNSAT cores are covered.

Deriving such an UNSAT core can be done by setting all the suspect variables in Equation 2.7 (from Section 2.6) to 0. Since the simulated outputs of the circuit do not match the expected outputs (or else there is no failure), the instance is unsatisfiable and an UNSAT core can be generated. However, in general, finding multiple UNSAT cores is difficult. The work in [131] finds multiple disjoint UNSAT cores by removing previously found UNSAT cores and calling the SAT-solver again. However, this technique will not find overlapping UNSAT cores which is

\[1\] That is, minimum error cardinality solution, see Section 2.3.
a common situation. For example, it is quite likely that multiple errors propagate and converge to a single point near the mismatched output pin. This means that the applicability of the technique is most useful when multiple UNSAT cores, overlapping or non-overlapping, are found. Unfortunately, finding non-overlapping UNSAT cores requires significant computation [89, 108, 110] in general. In practice, a single UNSAT core can be generated easily while multiple disjoint cores can be found if the problem happens to contain them.

The following example demonstrates the use of a single UNSAT core to restrict the solution space for finding a single design error.

**Example 23** Figure 5.2 shows a visualization of the debug instance from Equation 2.4. A simple combinational circuit has been augmented with the error model denoted by ⊗ which disconnects a gate’s fan-out from its fan-in to become free. Next, constraints are added for the input and expected outputs. Notice that the outputs mismatch on $g_4$ and $g_5$.

By disabling all suspect variables, we can generate an UNSAT core involving variables $e_1$, $e_2$ and $e_4$ for this instance. This core implies if there is a single design error, it must be among the corresponding gates. When solving the SAT-instance for a single design error ($N = 1$), $e_2 = 1$ is the only satisfiable solution corresponding to gate $g_2$, confirming the information found in the UNSAT core.
5.3 Efficient Debugging of Multiple Design Errors

Unsatisfiable cores can provide valuable information in determining which locations are involved in the observed failure. However, as mentioned previously, finding multiple cores is not a simple task. In this section, we describe a method for finding a single unsatisfiable core at each error cardinality that can be used to reduce the search space for higher cardinalities. The unsatisfiable cores generated are a natural result of the solving process at each cardinality and fit within existing SAT-based design debugging frameworks.

5.3.1 Reducing the Search Space of Multiple Design Errors

As previously described, an unsatisfiable core intuitively contains information regarding which parts of the design are responsible for the observed failure. Given a single core that is derived by setting all the suspect variables to 0, we know that at least one of the suspect variables in the core must be activated in order to generate a satisfying assignment. However, this result can be generalized to higher cardinalities as well.

When debugging at a certain cardinality \( N \), one finds all possible satisfying solutions by adding blocking clauses. Once the solver proves unsatisfiability at that cardinality, it also generates an unsatisfiable core. However, intuitively, this core differs from the one described in Section 5.2.1. Instead of only finding a single conflict, the suspect variables allow additional flexibility in breaking up to \( N \) conflicts. However, even by breaking \( N \) conflicts, there still must exist at least one more conflict that cannot be broken. To achieve a SAT result, we must be able to break each one of these conflicts. This means that some subset of suspect variables (\( > N \)) involved in these conflicts are needed in order to break the resulting core. This leads to the result that if a suspect variable is not included in the unsatisfiable core at \( N \), then it cannot be in a solution at \( N + 1 \). This is stated more precisely in the next theorem.

**Theorem 3** Let \( U \) be the unsatisfiable core generated by solving and blocking all solutions for the debug instance \( \text{Debug}_{0}^{k}(N) \). If \( e_i \notin U \), then \( e_i = 1 \) is not in any satisfying assignment of the instance \( \text{Debug}_{0}^{k}(N + 1) \) with all lower cardinality solutions blocked.

*Proof:* Assume towards a contradiction that a suspect variable \( e_i \notin U \) and \( e_i = 1 \) is part of
a satisfying assignment for $\text{Debug}^k_0(N+1)$. Since $e_i = 1$, then exactly $N$ other suspect variables may be active. $U$ is an unsatisfiable core that does not contain $e_i$ and cannot be broken with $N$ or less suspect variables being active. Therefore, $\text{Debug}^k_0(N+1)$ is unsatisfiable, leading to a contradiction. So it must be the case that $e_i = 1$ is not part of any satisfying solution for $\text{Debug}^k_0(N+1)$.

Although Theorem 3 gives us valuable information about the next cardinality, it can also provide information about higher cardinalities. The core can help us constrain the solution space of higher cardinalities by specifying a superset of the suspect variables which must be active. This is described in the next corollary.

**Corollary 1** Given $U$ from Theorem 3, any solution found at a cardinality greater than $N$ must have at least $N+1$ suspect variables active from $U$.

**Proof:** If fewer than $N+1$ suspect variables from $U$ are active, then $U$ still forms an UNSAT core so it cannot be part of any solution for a higher cardinality. Therefore, any satisfying solution must involve at least $N+1$ suspect variables from $U$. 

Notice that the results in [131] are a special case of Theorem 3 and Corollary 1 where $N = 0$. Similar results for multiple UNSAT cores can be applied for the higher cardinality cores as well. However, as mentioned before, these multiple cores are difficult to generate efficiently. The next example shows how Theorem 3 helps reduce the search space at higher cardinalities.

**Example 24** Continuing from Example 23, if we use the UNSAT core resulting from debugging instance with $N = 1$, we will get an UNSAT core involving variables $e_1$, $e_2$, $e_4$ and $e_5$. When solving the $N = 2$ instance (while blocking previously found solutions), we will get a single solution of $e_4 = 1 \land e_5 = 1$ corresponding to gates $g_4$ and $g_5$. This confirms the result from Theorem 3 where the suspects in the solution must be in the UNSAT core from the previous cardinality.

Since unsatisfiable cores are generated at each cardinality as a by-product of solving process, Theorem 3 can be used effectively with very little overhead. An integrated algorithm is described in the next sub-section.
5.3.2 Overall Algorithm

Algorithm 5.1 Debugging Multiple Design Errors

1: \( N_{\text{max}} := \) error cardinality
2: \( \text{sols} := \) solutions found by algorithm
3: procedure \( \text{DebugMultipleErrors}(N_{\text{max}}) \)
4: \( U \leftarrow \{\}, \text{sols} \leftarrow \{\} \)
5: for \( n : 0 \ldots N_{\text{max}} \) do
6: \( \text{inst}_{\text{orig}} \leftarrow \text{Debug}(n) \)
7: \( \text{inst}_{\text{opt}} \leftarrow \text{DISMISSSuspects}(\text{inst}_{\text{orig}}, U) \)
8: \( \text{sols} \leftarrow \text{sols} \cup \text{SOLVEALL}(\text{inst}_{\text{opt}}) \)
9: \( U \leftarrow \text{EXTRACTCore}(\text{inst}_{\text{orig}} \land \text{block}(\text{sols})) \)
10: end for
11: return \( \text{sols} \)
12: end procedure

Algorithm 5.1 presents pseudo-code for the overall algorithm to debug multiple design errors. The algorithm takes as input the maximum error cardinality \( (N_{\text{max}}) \) and outputs solutions corresponding to possible locations for the design error(s). Lines 5-10 consist of the main loop where the debugging instance is solved for each error cardinality and an UNSAT core is generated. First, the debugging instance is created at the current error cardinality on line 6. Next, Theorem 3 is applied where suspects not inside the previously found core are removed from the instance (line 7). This can either be done by re-generating the instance or simply setting the corresponding suspect variables to 0. Using this optimized instance all solutions are found (line 8). Finally, the UNSAT core is extracted on line 9.

On line 9, it is important to use the unoptimized instance \( (\text{inst}_{\text{orig}}) \) that does not contain the blocked suspects to derive the current UNSAT core. This is due to the instance generated on line 7 removing certain suspect variables from consideration. These removed variables apply only to the current cardinality, not higher ones. If the optimized instance \( (\text{inst}_{\text{opt}}) \) were used instead, it would generate a core which incorrectly constrain the solution space.
A more general method could use Corollary 1 instead. In this way, additional constraints could be used to force suspect variables in the core to be at least a certain number. However, too many additional clauses would be needed to model this constraint, negating the benefit of the optimization in the first place. Thus, in practice Theorem 3 provides a greater benefit over Corollary 1.

5.4 Experiments

In this section we present results for the proposed algorithm for debugging multiple design errors. The same general experimental setup as Chapter 3 and Chapter 4 is used for this chapter with only several parameters changed. All experiments are performed using a single core of an Intel Core i5 3.1 GHz machine with memory limit of 8GB and a timeout of 7200 seconds. The debugger used is a C++ sequential SAT-based engine based on MiniSAT. MiniSAT is used to solve all the SAT instances with a commercial Verilog frontend to allow for diagnosis of RTL designs. The UNSAT core extraction feature in the solver is turned on only during the ExtractCore step in Algorithm 5.1. All other runs used the default solver settings with the core extraction feature off.

The effectiveness of the proposed technique is shown on a variety of RTL designs from OpenCores. Each debug instance is generated by randomly selecting a line in the RTL and inserting a typical RTL design error such as an incorrect operator, state transition, or module instantiation. Note that these RTL errors typically translate to multiple design errors at the gate level. Suspect variables are then placed on each gate of the synthesized design that correspond to an actual signal in the RTL. Next, each instance is run through its accompanying testbench, simulated and the resulting error trace is recorded. The instances are labeled with the circuit name followed by a number indicating different errors that were inserted. The designs used in these experiments overlap with designs used in other chapters, but may differ in their design characteristics. This difference can attributed to the version of the design and the version of the Verilog frontend.

The experimental results for Algorithm 5.1 with $N_{max} = 3$ are shown in Table 5.1 and...
Table 5.1: Instance information for experimental results showing the design characteristics, cycles analyzed, and number of solutions.

<table>
<thead>
<tr>
<th>instance</th>
<th># gates</th>
<th># flipflops</th>
<th># cycles analyzed</th>
<th># suspect locations</th>
<th># solutions (N=1)</th>
<th># solutions (N=2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac97ctrl1</td>
<td>15109</td>
<td>2482</td>
<td>300</td>
<td>2483</td>
<td>25</td>
<td>3</td>
</tr>
<tr>
<td>ac97ctrl2</td>
<td>15114</td>
<td>2483</td>
<td>200</td>
<td>2456</td>
<td>12</td>
<td>132</td>
</tr>
<tr>
<td>divider1</td>
<td>3773</td>
<td>424</td>
<td>39</td>
<td>498</td>
<td>18</td>
<td>503</td>
</tr>
<tr>
<td>memctrl1</td>
<td>46767</td>
<td>1239</td>
<td>40</td>
<td>2858</td>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>mriscl1</td>
<td>15407</td>
<td>2161</td>
<td>41</td>
<td>2566</td>
<td>52</td>
<td>118</td>
</tr>
<tr>
<td>mriscl2</td>
<td>14456</td>
<td>1371</td>
<td>40</td>
<td>2044</td>
<td>61</td>
<td>358</td>
</tr>
<tr>
<td>rsdecoder1</td>
<td>13023</td>
<td>526</td>
<td>40</td>
<td>5159</td>
<td>21</td>
<td>102</td>
</tr>
<tr>
<td>usbfunct1</td>
<td>10217</td>
<td>736</td>
<td>37</td>
<td>1593</td>
<td>27</td>
<td>223</td>
</tr>
<tr>
<td>vga1</td>
<td>72292</td>
<td>17110</td>
<td>50</td>
<td>2936</td>
<td>38</td>
<td>328</td>
</tr>
<tr>
<td>vga2</td>
<td>73546</td>
<td>17213</td>
<td>100</td>
<td>1103</td>
<td>118</td>
<td>116</td>
</tr>
</tbody>
</table>
Table 5.2: Experiments for the proposed DEBUGMULTEPLEERRORS algorithm (Algorithm 5.1) compared against the basic formulation of SAT-based debugging [128].

<table>
<thead>
<tr>
<th>instance info</th>
<th>SAT Debug [128]</th>
<th>Proposed DEBUGMULTEPLEERRORS Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time (s)</td>
<td>peak memory (MB)</td>
</tr>
<tr>
<td>ac97ctrl1</td>
<td>294</td>
<td>2466</td>
</tr>
<tr>
<td>ac97ctrl2</td>
<td>402</td>
<td>1937</td>
</tr>
<tr>
<td>divider1</td>
<td>653</td>
<td>487</td>
</tr>
<tr>
<td>memctrl1</td>
<td>57</td>
<td>1309</td>
</tr>
<tr>
<td>mirc1</td>
<td>317</td>
<td>913</td>
</tr>
<tr>
<td>mirc2</td>
<td>637</td>
<td>708</td>
</tr>
<tr>
<td>rsdecoder1</td>
<td>2518</td>
<td>1000</td>
</tr>
<tr>
<td>usbfunct1</td>
<td>1298</td>
<td>788</td>
</tr>
<tr>
<td>vga1</td>
<td>3958</td>
<td>1639</td>
</tr>
<tr>
<td>vga2</td>
<td>57</td>
<td>1119</td>
</tr>
</tbody>
</table>
Table 5.2. The proposed algorithm is denoted by \textit{core} and compared against the SAT-based debugger described in \cite{128}, denoted by \textit{orig}. Each row of the tables correspond to a different instance that is run.

In Table 5.1, the first five columns of the table show the instance name, number of combinational gates, number of state elements, length of the error trace, and number of potential suspect locations. The next three columns show the number of solutions for each of the three error cardinalities which are identical for both \textit{core} and \textit{orig}.

In Table 5.2, the first two columns show the run-time in seconds and the peak memory in MB for \textit{orig}. The next six show the results for \textit{core}. These columns show the cumulative time needed to generate all the UNSAT cores, total run-time including core generation, peak memory and percentage of suspects removed from consideration for each of the three error cardinalities.

The benefit of the proposed technique is clearly shown when looking at the total run-time compared with the previous work. There is an average reduction 22\% in the total run-time when using UNSAT cores (using a geometric mean, a 1.31x speedup on average). This comes with almost no impact to peak memory, resulting in a slight reduction of 0.4\% in favor of the \textit{core} technique. This improvement in run-time can be primarily explained by the drastic reduction in the search space for each cardinality. From columns 7-9 in Table 5.2, the average reduction in terms of number of suspects across all instances are 83\%, 74\% and 66\% for $N = 1$ to $N = 3$ respectively.

Figure 5.3 shows the number of dismissed suspects for several instances for each $N$. When the number of dismissed suspects is large, there is significant benefit to total run-time, as in the case of \texttt{vga} with a run-time reduction of 34\%. However, when the number of dismissed suspects is small, as in the case of \texttt{divider1}, the benefit of reducing the solve time may not outweigh the increased overhead of finding an UNSAT core. A similar case occurs in \texttt{usbfunct1} where the trade-off of finding a core at $N = 2$ did not pay off during the solving of $N = 3$. Although these two instances show negative results, overall the technique still performs well in reducing run-time.

In terms of peak memory, the two sets of experiments returned mixed results. In general, extracting an UNSAT core causes an increase in memory usage due to the necessary bookkeeping
needed to keep track of the core during solving. However, for these experiments the peak memory occurred primarily while solving the $N = 3$ case where core extraction is turned off. As a result, the memory results are almost identical to each other.

### 5.5 Summary

This chapter presents a new algorithm for debugging multiple design errors efficiently. It builds upon previous work by generalizing the concept of how to generate as well as apply unsatisfiable cores to reduce the solution space of the debugging problem. The generation and application of these cores fit naturally within existing SAT-based debugging frameworks to allow easy extensibility of existing implementations. Experimental results show significant improvements to run-time while having minimal impact on peak memory.
Chapter 6

Automated Debugging of Assertions

6.1 Introduction

Modern assertion-based verification (ABV) environments are typically composed of three main components: design, testbench and assertions. Due to the human factor inherent in the design process, it is equally likely for errors to be introduced into any one of these components. Commercial solutions [66, 116, 127] aim to help the debugging process by allowing manual navigation and visualization of these components. Most existing research in automated debugging [27, 55, 67, 128, 137] have focused primarily on locating errors within the design. The absence of automated debugging tools targeting testbenches and assertions remains a critical roadblock in further reducing the verification bottleneck.

The adoption of assertions introduces new challenges to the debugging process. Modern temporal assertion languages such as SystemVerilog assertions and Property Specification Language [1, 3] are foreign to most design engineers who are more familiar with RTL semantics. Temporal assertions concisely define behaviors across multiple cycles and execution threads, which creates a significant paradigm shift from RTL. For example, debugging the failing SystemVerilog assertion req |= gnt ##[1:4] ack requires the engineer to analyze four threads over five clock cycles to understand the failure. Moreover, a single temporal operator such as a non-consecutive repetition may map to a multiple line RTL implementation, adding to the debugging complexity. For these reasons, debugging complex temporal assertions remains one
Chapter 6. Automated Debugging of Assertions

of the biggest challenges in their widespread adoption.

Automated circuit debugging techniques have traditionally relied on localizing an error in a circuit. In a similar manner, it is possible to synthesize assertions \[21, 41, 93\] and allow one to apply similar circuit localization techniques \[128\] from automated design debugging to assertions. However, this proves ineffective in debugging assertions due to their compact nature, also shown later in this chapter. For example, applying path-tracing \[67\] to the assertion valid

\[
\text{##1 start } \Rightarrow \text{ go}
\]

will return the entire assertion as potentially erroneous. Moreover, this type of localization does not provide help in directing the engineer towards correcting it. This suggests an urgent need for a departure from traditional circuit debugging techniques so that we can debug assertions effectively.

In this chapter, we propose a novel automated debugging methodology for SystemVerilog assertions (SVA) that takes a different approach. It aids debugging by generating a set of properties closely related to the original failing assertion, where each generated property returned is guaranteed to be validated against the RTL. These properties serve as a basis for possible corrections to the failing assertion, providing an intuitive method for debugging and correction. They also provide insight into design behavior by being able to contrast their differences with the failing assertion. In summary, our major contributions are as follows:

- We introduce a language independent methodology for debugging errors in assertions that produce a set of closely related verified properties to aid the debugging process. These properties are generated by an input set of modifications that mutate existing assertions.

- We propose a particular set of modifications for the SystemVerilog assertion language to mutate the failing assertion in order to generate closely related properties.

- We introduce two techniques dealing with vacuity and multiple errors to enhance the practical viability of this approach.

An extensive set of experimental results are presented on real designs with SystemVerilog assertions written from their specifications. They show that the proposed methodology and modification model are able to return high quality verified properties for all instances. In
addition, the multiple error and vacuity techniques are able to filter out inessential properties by an average of 23% and 34%, respectively.

The remaining sections of the chapter are organized as follows. Section 6.2 discusses related work. Our contributions are presented in Section 6.3-6.5. Section 6.6 presents the experimental results and Section 6.7 concludes this work.

6.2 Related Work

Mutation analysis (or mutation testing) [76] has been widely used to evaluate the quality of software or hardware tests. It works by making small modifications to the source code and determining if a test suite is able to detect the injected modification. This general methodology has been used in a hardware context to evaluate the quality of both testbenches and formal properties [42, 64]. A particular example of mutation analysis is in this industrial tool [134], which measures the quality of the verification environment and coverage holes. However, none of these solutions explicitly targets debugging functional errors within the assertion.

6.3 Assertion Debugging Methodology

This section presents a methodology that automatically debugs errors in failing assertions. It is assumed that errors only exist in the assertions and the design is implemented correctly. This parallels the assumption in automated RTL debuggers [27, 55, 67, 128, 137] where the verification environment is assumed to be correct. However, even with this assumption, the flow can still be realistically used by running an RTL debugger in conjunction with the proposed technique and returning the union of the two results. Using these results, the engineer can then determine where to make the appropriate fix. As an added benefit, the results from the assertion debugging methodology can also provide insight into the design behavior to aid both assertion and RTL debugging. Note that this methodology makes no assumptions about the assertion language or the types of errors as these are functions of the input model.

The methodology aids debugging by returning a set of verified properties with respect to the design that closely relate to the failing assertion. We denote this set as $P$. This set of
closely related assertions aids in the debugging process in several ways. First, $P$ serves as a suggestion for possible corrections to the failing assertion. As such, it provides an intuitive method to aid in the debugging and correction process. Second, since the properties in $P$ have been verified, they provide an in-depth understanding of related design behaviors. This information is essential in understanding the reason for the failed assertion. Finally, $P$ allows the engineer to contrast the failing assertion with closely related ones, a fact that allows the user to build intuition regarding the possible sources and causes of errors.

The overall methodology is shown in Figure 6.1 and consists of three main steps. After a failing assertion is detected by verification (either testbench or formal verification), the first step of applying mutations is performed. This step takes in the failing property along with the mutation model and generates a set of closely related properties, denoted as $P'$, to be
verified. Each property in $P'$ is generated by taking the original failing assertion and applying one or more pre-defined modifications, or \textit{mutations}, defined by the mutation model. This model defines the ability of the methodology to handle different assertion languages as well as different types of errors. We define a practical model for SVA in the next section but different models based on user experience are also possible.

The second step of the methodology quickly rules out invalid properties in $P'$ through simulation with the failing counter-example\textsuperscript{1}. A counter-example in this context is a simulation trace that shows one way for the assertion to fail. The intuition here is that since the counter-example causes the original assertion to fail, it will also provide a quick filter for related properties in $P'$. This is accomplished by evaluating each property in $P'$ for each cycle in the counter-example through simulation. If any of the properties in $P'$ fail for any of the evaluations, they are removed from $P'$. The resulting set of properties is denoted by $P''$.

The final step of the methodology uses an existing verification flow to filter out the remaining invalid properties in $P''$. This is the most time-consuming step in this process, which is the reason for generating the initial filtered set $P''$. The existing verification flow can either be a high coverage simulation testbench or a formal property checker. In the case of the testbench, the properties in $P$ will have a high confidence of being true, while with the formal flow, $P$ is a set of proven properties for the design. In most verification environments, both flows are automated resulting in no wasted manual engineering time. The final set of filtered properties $P$ are verified by the environment and can be presented to the user for analysis.

\section{SystemVerilog Assertion Mutation Model}

This section describes a practical mutation model for SystemVerilog assertions to be used with the methodology described in Section 6.3. These mutations are designed to model common industrial errors \cite{57,132} as well as misinterpretations of SVA \cite{140}. This model is created based on our discussions with industrial partners, our experiences writing assertions, as well as

\footnote{As noted in Section 2.2.2, error trace and counter-example are functionally synonymous terms with respect to debugging. However, since this Chapter deals specifically with assertions, we will use the term counter-example instead of error trace due to its prevalence in assertion literature.}
common errors cited in literature. Note that other mutation models can be developed based on user experience.

Each mutation modifies the assertion by either adding operators, changing operators or changing parameters to operators. Each new property is generated by applying a fixed number of these mutations to the failing assertion. The number of mutations is defined to be the cardinality of the candidate and depends on the number of additions or changes to the assertion. In some cases, multiple or complex errors may require higher cardinality to model. The rest of the section will describe the various types of mutations in this model.

The first group of mutations involves modifying Boolean expressions. SVA provides operators for signal transitions across a pair of clock cycles and previous values. These operators can frequently be misused. For example, a common error occurs when interpreting the word “active”. It is ambiguous whether the intent is a rising edge ($\text{rose}(\text{sig})$), or a level sensitive trigger ($\text{sig}$). Similarly with the $\text{past}(\text{sig},k)$ operator, the number of cycles ($k$) to evaluate in the past is a common source of errors. The mutation can model this error by adding or subtracting an integer $i$. Table 6.1 gives the mapping from Boolean operators to the set of possible mutations where $\langle s \rangle$ is a given signal.

<table>
<thead>
<tr>
<th>Name</th>
<th>Operator/Expression</th>
<th>Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean Expressions</td>
<td>$\langle s \rangle$, $\text{past}$, $\text{rose}$, $\text{fell}$, $\text{stable}$, $\text{changed}$</td>
<td>Replace with ${ \langle s \rangle$, $!\langle s \rangle$,$\text{rose}(\langle s \rangle)$, $\text{past}(\langle s \rangle$, $k \pm i)$, $\text{fell}(\langle s \rangle)$, $\text{stable}(\langle s \rangle)$, $\text{changed}(\langle s \rangle)}$</td>
</tr>
</tbody>
</table>

The next group of mutations involve commonly used arithmetic, logical and bitwise operators. They replace such operators with their related counter-parts. For example, if $\&\&$ is mutated into $\|$, it may relax a condition to generate a passing property. This can provide insight into possible places where the error may have occurred. Another example would be replacing $>$ with $\geq$ which might also correct a commonly made error. The mutations for these types are operators are shown in Table 6.2.
Table 6.2: Arithmetic, Logic and Bitwise Mutations

<table>
<thead>
<tr>
<th>Name</th>
<th>Operator/Expression</th>
<th>Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical</td>
<td>!, &amp;&amp;,</td>
<td></td>
</tr>
<tr>
<td>Bitwise</td>
<td>¬, &amp;,</td>
<td>, ∧</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>+, -</td>
<td>Replace with {+, -}</td>
</tr>
<tr>
<td>Equality</td>
<td>&lt;, &lt;=, ==, &gt;=, &gt;, !=</td>
<td>Replace with {&lt;, &lt;=, ==, &gt;=, &gt;, !=}</td>
</tr>
</tbody>
</table>

SVA sequential binary operators comprise the next group of mutations, which have subtle differences that are easy to misinterpret. For example, intersect is similar to and except with the added restriction that the lengths of the matched sequences must be the same. Another example is replacing the intersect operator with the within operator, which might produce a passing property that can give insight into the underlying error. Table 6.3 shows the possible mutations.

Table 6.3: Sequential Binary Mutations

<table>
<thead>
<tr>
<th>Name</th>
<th>Operator/Expression</th>
<th>Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>and, intersect, or, within</td>
<td>Replace with {and, intersect, or, within}</td>
</tr>
</tbody>
</table>

The next group of mutations involve the sequential concatenation operator. This family of operators specifies a delay and is frequently used in properties. Two types of delays are possible – a single delay or a range of delays. The mutations involve changing the delays by integers \(i\) or \(j\), or changing a single delay into a ranged delay. When mutating using \(i\) or \(j\), the cardinality will be increased by the absolute value of the integer. For example, changing \(##1\) to \(##3\) will increase the cardinality by 2. Table 6.4 describes these mutations.

Sequential repetition operators are the next group of mutations. These operators allow repetition of signals in consecutive or non-consecutive forms with subtle differences. For example, the non-consecutive goto repetition \(\text{sig}[-k_1]\) is frequently confused with the \(\text{sig} [= k_1]\) operator. The difference between the two is whether the sequence ends with strictly \(k_1\) matches, or if multiple cycles are allowed before the next occurrence. The mutations may either change
Table 6.4: Sequential Concatenation Mutations

<table>
<thead>
<tr>
<th>Name</th>
<th>Operator/Expression</th>
<th>Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential Concatenation</td>
<td>##k_1,##[k_1:k_2]</td>
<td>Change single delay to range delay; Change constants to $k_1 \pm i, k_2 \pm j$</td>
</tr>
</tbody>
</table>

the repetition operator or the number of repetitions performed using $i$ and $j$. Table 6.5 shows these mutations.

Table 6.5: Sequential Repetition Mutations

<table>
<thead>
<tr>
<th>Name</th>
<th>Operator/Expression</th>
<th>Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repetition</td>
<td>[*k_1],[*k_1:k_2]</td>
<td>Replace op with {[*],[=],[-&gt;]}; Change constant to $k_1 \pm i, k_2 \pm j$</td>
</tr>
<tr>
<td></td>
<td>[=k_1],[=k_1:k_2]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[-&gt;k_1],[-&gt;k_1:k_2]</td>
<td></td>
</tr>
</tbody>
</table>

The last group of mutations involve the implication operators. This family of operators are often used because most properties are evaluations based on a condition. The first type of mutation is a change between the overlapping ($|->$) and non-overlapping ($|=->$) implication. This accounts for when there is extra or missing delay between the antecedent and consequent. The next mutation extends this idea by allowing a multiple cycle delay after the antecedent with the addition of the $$i operator. The third type of mutation in this group involves adding the first match operator to the antecedent of the implication. This addresses a subtlety of SVA where the consequent of each matching antecedent thread must be satisfied. The first match operator handles this subtlety by allowing only the first matching sequence to be used. Table 6.6 outlines the possible mutations.

The following example shows how mutations can be applied in the context of an entire property.

**Example 25** Consider the assertion described in Example 11 (from Section 2.7). We can generate properties in $P'$ by mutating the failing assertion with each of the applicable rules. Here we show several examples of mutated properties with different cardinalities.
Table 6.6: Implication Mutations

<table>
<thead>
<tr>
<th>Name</th>
<th>Operator/Expression</th>
<th>Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implication</td>
<td></td>
<td>=&gt;,</td>
</tr>
</tbody>
</table>

// Cardinality 1
// If start high, then one cycle later, data high needs to repeat non-consecutively twice, followed by stop high two cycles later, followed by stop low one cycle later.

P1: start |=> data[->2] ##2 stop ##1 !stop

// If start high, then in the same cycle, data high needs to repeat non-consecutively twice, followed by stop high one cycle later, followed by stop low one cycle later.

P2: start |-> data[->2] ##1 stop ##1 !stop

// Cardinality 2
// If rising edge of start, then in the next cycle, data high needs to repeat non-consecutively twice followed by any number of cycles where data is low, followed by stop high one cycle later, followed by stop low one cycle later.

P3:$rose(start)|=> data[=2] ##1 stop ##1 !stop

// If rising edge of start, then in the next cycle, data high needs to repeat non-consecutively twice, followed by stop high two cycles later, followed by stop low one cycle later.

P4:$rose(start)|=> data[->2] ##2 stop ##1 !stop

// Cardinality 3
// If start is high, then in the next cycle, data low needs to repeat non-consecutively twice, followed by stop high one cycle later, followed by stop low three cycles later.

P5:start |=> !$data[->2] ##1 stop ##3 !stop
6.5 Practical Considerations and Extensions

The methodology outlined in Section 6.3 along with the model in Section 6.4 generates a set of closely related properties, \( P \). However, they are only useful in practice if the number of properties returned by the methodology is small enough to be analyzed by an engineer. Two techniques that greatly reduce the number of properties are discussed here.

The first technique deals with *vacuous assertions*. Assertions that are vacuous typically are considered erroneous since their intended behavior is not exercised. Similarly, all verified properties that are found to be vacuous for all evaluations are removed from \( P \). This reduces its size significantly as seen in the experimental results.

The second technique addresses *multiple cardinalities*. As the cardinality increases, the size of the mutated properties, \( P' \), increases exponentially. This may become unmanageable at higher cardinalities. To deal with this, \( P' \) can be reduced by eliminating properties with mutations that have been verified at lower cardinalities. For example if the property \( P_1 \) from Example 25 is found to be a verified property, it would remove \( P_4 \) from consideration since it contains the same mutation from \( P_1 \). The intuition here is that the removed properties do not add value because they are more difficult to contrast with the original assertion. This proves to be very effective in reducing the size of \( P' \) for higher cardinalities by removing these inessential properties.

6.6 Experiments

This section presents the experimental results for the proposed work. All experiments are run on a single core of a dual-core AMD Athlon 64 4800+ CPU with 4 GB of RAM. A commercial simulator [101] or property checker [102] is used for all simulation and verification steps. All instances are generated from Verilog RTL designs from OpenCores [111] and our industrial partners. SVA is written for all designs based on their specifications. A commercial SystemVerilog frontend [139] is used to parse and elaborate all designs and assertions in this chapter. The designs in this chapter may be used in experiments for other chapters, but may differ in their design characteristics. This difference can attributed to the version of the design and the
version of the frontend.

To generate unbiased results, we do not artificially insert errors directly into the assertions and then try to fix them. Instead, we add errors into the RTL to create a mismatch between the RTL implementation and SVA. We then assume that the RTL is correct and the SVA is erroneous, to create a failing assertion. It should be noted that in some cases there may be no possible corrections to the SVA since the RTL error may drastically change the design behavior.

The RTL errors that are injected are based on the extensive experience from our industrial partners. These are common design mistakes such as a wrong state transition, incorrect operator or incorrect module instantiation. It should be emphasized that RTL errors typically correspond to multiple gate-level errors.

To create instances for the experiments, for each RTL error, one assertion is selected as the mutation target among the failing assertions for a design. Each instance is named by appending a number after the design name. Table 6.7 shows the information for each of these instances. The columns of this table show the instance name, the number of gates, the number of state elements, followed by the number of operators plus variables in the assertion.

The following subsections present two sets of experiments. The first subsection demonstrates the ineffectiveness of circuit based localization techniques in debugging assertions, motivating the proposed methodology. The second subsection presents the experimental results from the proposed assertion debugging methodology.

### 6.6.1 Localization

To motivate and illustrate the impact of the proposed methodology, results of a circuit localization technique applied to debugging assertions are presented in this subsection. The instances from Table 6.7 that had simulation testbenches are used in these experiments. A path tracing strategy is implemented to locate which operators or variables could be responsible for an error in the assertion. This was done by replacing variables or nodes with any constant values and evaluating if the property passed. The counter-example used in these experiments is generated from its simulation testbench.

The results of these experiments are presented in Table 6.8. The columns of this table show
Table 6.7: Instance information for experiments showing the number of gates, number of flipflops, and number of SVA operators and variables for each corresponding assertion.

<table>
<thead>
<tr>
<th>instance</th>
<th># gates (k)</th>
<th># flipflops</th>
<th># SVA operators and variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>hp1</td>
<td>20.3</td>
<td>2164</td>
<td>12</td>
</tr>
<tr>
<td>hp2</td>
<td>20.3</td>
<td>2164</td>
<td>16</td>
</tr>
<tr>
<td>mips1</td>
<td>30.1</td>
<td>2670</td>
<td>19</td>
</tr>
<tr>
<td>mips2</td>
<td>30.1</td>
<td>2670</td>
<td>9</td>
</tr>
<tr>
<td>risc1</td>
<td>15.8</td>
<td>1371</td>
<td>14</td>
</tr>
<tr>
<td>risc2</td>
<td>15.8</td>
<td>1371</td>
<td>16</td>
</tr>
<tr>
<td>spi1</td>
<td>1.6</td>
<td>132</td>
<td>12</td>
</tr>
<tr>
<td>spi2</td>
<td>1.6</td>
<td>132</td>
<td>12</td>
</tr>
<tr>
<td>tlc1</td>
<td>2.5</td>
<td>42</td>
<td>7</td>
</tr>
<tr>
<td>tlc2</td>
<td>2.5</td>
<td>42</td>
<td>14</td>
</tr>
<tr>
<td>usb1</td>
<td>39.2</td>
<td>2349</td>
<td>4</td>
</tr>
<tr>
<td>usb2</td>
<td>39.2</td>
<td>2349</td>
<td>16</td>
</tr>
<tr>
<td>vga1</td>
<td>89.4</td>
<td>17110</td>
<td>8</td>
</tr>
<tr>
<td>vga2</td>
<td>89.4</td>
<td>17110</td>
<td>8</td>
</tr>
<tr>
<td>wb1</td>
<td>5.2</td>
<td>96</td>
<td>6</td>
</tr>
<tr>
<td>wb2</td>
<td>5.2</td>
<td>96</td>
<td>7</td>
</tr>
</tbody>
</table>

From the table, we see that path tracing returns suspects covering a large part of the assertion for all instances. This ranges from 50% for wb1 to 100% for usb1. On average, 71% of the assertion operators and variables are returned, confirming the ineffectiveness of circuit debugging techniques for use with assertions. This motivates the need for the mutation based debugging methodology presented in this work.

6.6.2 Assertion Debugging Methodology

The experimental results from implementing the proposed methodology, mutation model and enhancements are presented in this section. For each instance, two sets of experiments are run. The first uses the accompanying simulation testbench to generate the initial counter-example and the final verification step. The counter-example is generated by stopping the simulation at the point of first failure, while the verification runs through the entire testbench. We refer to
Table 6.8: Experimental results showing circuit localization techniques applied to debugging errors within assertions.

<table>
<thead>
<tr>
<th>instance</th>
<th>operators returned</th>
<th>operators returned(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hp1</td>
<td>9</td>
<td>75%</td>
</tr>
<tr>
<td>hp2</td>
<td>13</td>
<td>81%</td>
</tr>
<tr>
<td>mips1</td>
<td>17</td>
<td>89%</td>
</tr>
<tr>
<td>mips2</td>
<td>7</td>
<td>78%</td>
</tr>
<tr>
<td>spi1</td>
<td>6</td>
<td>50%</td>
</tr>
<tr>
<td>spi2</td>
<td>6</td>
<td>50%</td>
</tr>
<tr>
<td>tlc1</td>
<td>6</td>
<td>86%</td>
</tr>
<tr>
<td>tlc2</td>
<td>7</td>
<td>50%</td>
</tr>
<tr>
<td>usb1</td>
<td>4</td>
<td>100%</td>
</tr>
<tr>
<td>usb2</td>
<td>11</td>
<td>69%</td>
</tr>
<tr>
<td>vga1</td>
<td>4</td>
<td>50%</td>
</tr>
<tr>
<td>vga2</td>
<td>6</td>
<td>75%</td>
</tr>
<tr>
<td>wb1</td>
<td>3</td>
<td>50%</td>
</tr>
<tr>
<td>wb2</td>
<td>6</td>
<td>86%</td>
</tr>
</tbody>
</table>

these experiments as testbench. The second set instead uses a formal property checker for these tasks and we refer to them as formal. Note that not all instances have both a testbench and formal environment, thus some entries will be not available. In addition, for the same instance, different environments may produce different results due to assumption constraints in formal which are not respected in testbench. Each instance is run through the methodology varying the cardinality from one to three. Table 6.9 and Table 6.10 show the results of the testbench and formal experiments, respectively, in parallel tables.

The first two columns of both Table 6.9 and Table 6.10 show the instance name and cardinality. The next eight columns for each table show the experiments. Columns three and four show the size of the initial candidates, $P'$, without and with the multiple cardinality enhancement from Section 6.5, respectively. Columns five to seven show the number of passing assertions from $P'$ after simulation with the failing counter-example ($P''$), the number of vacuous properties found in $P''$, and the final number of verified properties ($P$). Column eight shows the percent reduction of the cardinality and vacuity enhancements from the unoptimized $P'$. Column nine and ten show the counter-example simulation time followed by the total testbench or formal verification time. Run-times to create the mutated properties ($P'$) take less than two seconds.
and are not shown in the table. The columns of Table 6.11 present supplemental data regarding the number of clock cycles in the testbench counter-example, testbench final verification step and formal counter-example, respectively. Time-outs for each step of the methodology are set at 3600 seconds and are indicated by TO.

The applicability of the proposed technique is apparent when analyzing the final number of verified properties in $P$. Despite the large number of initial properties in $P'$, the methodology successfully filters properties to a manageable size. This is important for debugging since it becomes impractical to use when $P$ is too large. Moreover, we see that in the case of $formal$, most instances are able to return proven properties that precisely specify the behavior of the design.

The SVA mutation model also proves to be helpful in generating high quality properties that pass verification. For example, for $hp1$ one of the cardinality three properties is: $\texttt{rose(rd)} \&\& \texttt{tim\_cas} \#\#(1) \!\!\!\!\texttt{rd[\*4]} \rightarrow \texttt{rose(wr\_safe)}$. The mutation here changed the repetition operator from $[\*7]$ to $[\*4]$. This directly corresponds to the RTL error which changed the timing between the read and write phases. Showing a different case for $spi2$, a cardinality one solution is: $\texttt{wfre} \rightarrow \texttt{$stable(rfwe)$[-> 1]}$ within $\texttt{$rose(state==1)$ ##0 (state==0)[-> 1]}$. The mutation added $\texttt{$stable$}$ to $\texttt{rfwe}$. This gives insight into the design behavior where the error in the RTL is that $\texttt{rfwe}$ does not toggle in the correct state.

Figure 6.2 shows the size of each set of properties on a log-scale for several sample instances. From the figure, we see that the multiple cardinality technique from Section 6.5 helps to reduce the size of $P'$ from the original unopt $P'$. This results in an average reduction across all instances of 23%. Next, we see that simulation with the counter-example efficiently reduces the size of $P'$ to $P''$ by an average of 59% across all instances. This is critical to ensure that the run-time of the final verification step is minimized.

For column six in Table 6.9 and Table 6.10 we see that the number of vacuous solutions also contributes to the reduced size of $P$ from $P''$. As a percentage of $P''$, these verified vacuous properties represent an average of 34% of the set. In addition, the enhancements from Section 6.5, shown in column eight, reduce the size of the unoptimized properties in $P'$ by 27% across all instances.
Finally from column ten in Table 6.9 and Table 6.10, we see that the run-time for the counter-example simulation and final verification step increases with the number of properties. For cardinality one, this does not significantly impact performance. For higher cardinalities, this becomes more costly and causes time-out conditions in certain formal cases. This degraded run-time is the trade-off for being able to model more complicated types of errors. However, this can be avoided with more precise input mutation models so that a costly increase in cardinality is not needed.

6.7 Summary

In this chapter, we present a methodology, mutation model and additional techniques to automatically debug errors in SystemVerilog assertions. The methodology works by using the assertion, counter-example and mutation model to generate a set of alternative properties that have been verified against the design. These properties serve as a basis for possible corrections and also provide insight into the design behavior and failing assertion. Experimental results show the methodology is effective in generating high quality alternative properties for all empirical instances.
<table>
<thead>
<tr>
<th>Instance</th>
<th>num. Mutations (N)</th>
<th># candidates ((P'))</th>
<th># candidates with cardinality (P')</th>
<th># candidates after simulation with counterexample (P'')</th>
<th># candidates returned (P)</th>
<th>% reduced with cardinality and vacuity (P)</th>
<th>counterexample sim. time (s)</th>
<th>full testbench sim. time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hp1</td>
<td>1</td>
<td>34</td>
<td>34</td>
<td>11</td>
<td>8</td>
<td>2</td>
<td>24%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>529</td>
<td>467</td>
<td>220</td>
<td>177</td>
<td>0</td>
<td>45%</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>5001</td>
<td>4129</td>
<td>2415</td>
<td>1824</td>
<td>38</td>
<td>54%</td>
<td>7</td>
</tr>
<tr>
<td>hp2</td>
<td>1</td>
<td>52</td>
<td>52</td>
<td>17</td>
<td>11</td>
<td>3</td>
<td>21%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1257</td>
<td>1116</td>
<td>441</td>
<td>334</td>
<td>2</td>
<td>38%</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>18748</td>
<td>15596</td>
<td>6521</td>
<td>4771</td>
<td>11</td>
<td>42%</td>
<td>54</td>
</tr>
<tr>
<td>mips1</td>
<td>1</td>
<td>62</td>
<td>62</td>
<td>20</td>
<td>15</td>
<td>5</td>
<td>24%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1797</td>
<td>1514</td>
<td>627</td>
<td>611</td>
<td>12</td>
<td>50%</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>32338</td>
<td>24235</td>
<td>11811</td>
<td>11660</td>
<td>14</td>
<td>61%</td>
<td>84</td>
</tr>
<tr>
<td>mips2</td>
<td>1</td>
<td>35</td>
<td>35</td>
<td>7</td>
<td>7</td>
<td>0</td>
<td>20%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>539</td>
<td>539</td>
<td>181</td>
<td>173</td>
<td>3</td>
<td>32%</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4791</td>
<td>4714</td>
<td>1952</td>
<td>1846</td>
<td>52</td>
<td>40%</td>
<td>26</td>
</tr>
<tr>
<td>spi1</td>
<td>1</td>
<td>38</td>
<td>38</td>
<td>10</td>
<td>1</td>
<td>9</td>
<td>3%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>682</td>
<td>400</td>
<td>33</td>
<td>24</td>
<td>5</td>
<td>45%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7638</td>
<td>3339</td>
<td>372</td>
<td>267</td>
<td>52</td>
<td>60%</td>
<td>5</td>
</tr>
<tr>
<td>spi2</td>
<td>1</td>
<td>39</td>
<td>39</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>693</td>
<td>527</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>24%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7445</td>
<td>4951</td>
<td>38</td>
<td>0</td>
<td>0</td>
<td>33%</td>
<td>7</td>
</tr>
<tr>
<td>tlc1</td>
<td>1</td>
<td>26</td>
<td>26</td>
<td>9</td>
<td>3</td>
<td>4</td>
<td>12%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>285</td>
<td>203</td>
<td>67</td>
<td>40</td>
<td>10</td>
<td>43%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1706</td>
<td>896</td>
<td>250</td>
<td>201</td>
<td>0</td>
<td>59%</td>
<td>1</td>
</tr>
<tr>
<td>tlc2</td>
<td>1</td>
<td>42</td>
<td>42</td>
<td>13</td>
<td>1</td>
<td>2</td>
<td>2%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>803</td>
<td>727</td>
<td>315</td>
<td>81</td>
<td>6</td>
<td>20%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9250</td>
<td>7737</td>
<td>3383</td>
<td>1279</td>
<td>103</td>
<td>30%</td>
<td>14</td>
</tr>
<tr>
<td>usb1</td>
<td>1</td>
<td>13</td>
<td>13</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>65</td>
<td>38</td>
<td>11</td>
<td>0</td>
<td>4</td>
<td>42%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>163</td>
<td>62</td>
<td>28</td>
<td>0</td>
<td>6</td>
<td>62%</td>
<td>1</td>
</tr>
<tr>
<td>usb2</td>
<td>1</td>
<td>51</td>
<td>51</td>
<td>20</td>
<td>16</td>
<td>0</td>
<td>31%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1206</td>
<td>1206</td>
<td>705</td>
<td>592</td>
<td>6</td>
<td>49%</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>17553</td>
<td>17301</td>
<td>11736</td>
<td>10239</td>
<td>70</td>
<td>60%</td>
<td>32</td>
</tr>
<tr>
<td>vga1</td>
<td>1</td>
<td>19</td>
<td>19</td>
<td>6</td>
<td>0</td>
<td>4</td>
<td>0%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>151</td>
<td>90</td>
<td>24</td>
<td>0</td>
<td>0</td>
<td>40%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>655</td>
<td>278</td>
<td>96</td>
<td>0</td>
<td>9</td>
<td>58%</td>
<td>1</td>
</tr>
<tr>
<td>vga2</td>
<td>1</td>
<td>22</td>
<td>22</td>
<td>7</td>
<td>2</td>
<td>5</td>
<td>9%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>223</td>
<td>132</td>
<td>38</td>
<td>13</td>
<td>4</td>
<td>47%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1358</td>
<td>567</td>
<td>170</td>
<td>40</td>
<td>2</td>
<td>61%</td>
<td>1</td>
</tr>
<tr>
<td>wb1</td>
<td>1</td>
<td>25</td>
<td>25</td>
<td>7</td>
<td>5</td>
<td>1</td>
<td>20%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>260</td>
<td>240</td>
<td>98</td>
<td>85</td>
<td>3</td>
<td>40%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1446</td>
<td>1238</td>
<td>616</td>
<td>560</td>
<td>9</td>
<td>53%</td>
<td>2</td>
</tr>
<tr>
<td>wb2</td>
<td>1</td>
<td>22</td>
<td>22</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>18%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>205</td>
<td>167</td>
<td>50</td>
<td>47</td>
<td>0</td>
<td>41%</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1050</td>
<td>754</td>
<td>235</td>
<td>214</td>
<td>0</td>
<td>49%</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 6.9: Assertion debugging methodology results with testbench verification.
Table 6.10: Assertion debugging methodology results with formal verification.

<table>
<thead>
<tr>
<th>Instance Info</th>
<th>Formal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Instance</td>
</tr>
<tr>
<td>hp1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>hp2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>risc1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>risc2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>spi1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>spi2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>tlc1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>tlc2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>usb1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>usb2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>vga1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>vga2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>
Table 6.11: Supplemental trace length data of instances used in the experimental results.

<table>
<thead>
<tr>
<th>instance</th>
<th>testbench error trace length (cycles)</th>
<th>testbench total length (k cycles)</th>
<th>formal error trace length (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hp1</td>
<td>20479</td>
<td>27</td>
<td>47</td>
</tr>
<tr>
<td>hp2</td>
<td>20466</td>
<td>27</td>
<td>43</td>
</tr>
<tr>
<td>mips1</td>
<td>25</td>
<td>2</td>
<td>N/A</td>
</tr>
<tr>
<td>mips2</td>
<td>3393</td>
<td>12</td>
<td>N/A</td>
</tr>
<tr>
<td>risc1</td>
<td>N/A</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>risc2</td>
<td>N/A</td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>spi1</td>
<td>35</td>
<td>7</td>
<td>27</td>
</tr>
<tr>
<td>spi2</td>
<td>34</td>
<td>7</td>
<td>26</td>
</tr>
<tr>
<td>tlc1</td>
<td>29</td>
<td>0.5</td>
<td>15</td>
</tr>
<tr>
<td>tlc2</td>
<td>495</td>
<td>0.5</td>
<td>11</td>
</tr>
<tr>
<td>usb1</td>
<td>11</td>
<td>0.4</td>
<td>11</td>
</tr>
<tr>
<td>usb2</td>
<td>219</td>
<td>0.4</td>
<td>9</td>
</tr>
<tr>
<td>vga1</td>
<td>43</td>
<td>8377</td>
<td>7</td>
</tr>
<tr>
<td>vga2</td>
<td>45</td>
<td>8424</td>
<td>7</td>
</tr>
<tr>
<td>wb1</td>
<td>13</td>
<td>5</td>
<td>N/A</td>
</tr>
<tr>
<td>wb2</td>
<td>15</td>
<td>5</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Chapter 7

Automated Debugging of Missing Assumptions

7.1 Introduction

Formal property checkers\footnote{35} aim to increase verification efficiency by exhaustively verifying an assertion that encodes design intent. In the ideal case, if an assertion is violated, the formal tool returns a single counter-example allowing detection and debugging of corner case design bugs. However, as extensively documented in industry reports\footnote{98}, debugging formal counter-examples can be challenging, as the engineer does not have confidence whether the observed failure is due to a design bug, an incorrectly written assertion, or a missing assumption. Despite the need for designer intervention to determine the actual root-cause of the failure, previous work\footnote{55, 80} in debug automation has shown to be effective in aiding the designer in the former two cases. However, diagnosing missing assumptions still poses a significant bottleneck in formal verification flow as they can cause up to 50\% of the formal failures\footnote{98}.

Assumptions are necessary in formal verification as they model the design’s intended environment and ensure that actual RTL bugs can be detected. Nevertheless, debugging missing assumptions can be a challenging task since, unlike assertions, they are rarely explicitly documented. Instead, they are expressed implicitly by either the design specification or the functionality of adjacent design blocks. For the engineer, this can lead to a tedious “guess-
and-check” iterative debugging process, introducing many time-consuming calls to the formal tool. To alleviate this burden and allow formal technology to realize its full potential, the engineer today needs more debug automation to analyze the behavior of the counter-example and identify candidate missing assumptions.

This chapter presents an automated assumption debugging methodology to aid debugging of missing assumptions. This methodology automatically generates fixed cycle input assumptions in the form of SystemVerilog properties from a single failing formal counter-example\(^1\). The benefit of these generated assumptions for debugging is twofold. First, the assumptions are generated efficiently from the counter-example without the need to re-run the entire formal flow, thus providing feedback quickly in the debugging cycle. Further, the assumptions are in the form of simple properties that can aid debugging by either being directly used for the actual missing assumption(s), or indirectly used to give intuition about the failure. The key insight is that the engineer cannot be taken out of the debugging loop entirely. Instead, the algorithm aims to efficiently return easy to understand feedback to speed up the debugging of missing assumptions.

In detail, the major contributions of the work are as follows:

- An algorithm to generate multiple distinct counter-examples from a single assertion failure by iteratively extracting constraints from previous counter-examples and re-running the formal tool. As an added benefit, when a missing assumption is not the root-cause of the observed failure, the generated counter-examples can be used by the engineer to improve the resolution of existing automated tools when debugging incorrect assertions and/or RTL design errors.

- A model to generate candidate fixed cycle assumptions that aid debugging by serving as the actual missing assumption or indirectly providing intuition about the failure.

- A method to prune the list of generated assumptions by removing candidates that cannot prevent the failures seen in the given counter-examples.

---

\(^1\)As noted in Section 2.2.2, error trace and counter-example are functionally synonymous terms with respect to debugging. However, since this Chapter deals specifically with formal verification, we will use the term counter-example instead of error trace due to its prevalence in formal verification literature.
An extensive set of experimental results have been performed over a wide variety of OpenCores designs with SystemVerilog assertions written from their specification documents. Multiple counter-examples are shown to reduce the number of generated assumptions by 38% on average for ten counter-examples, and an average of 28 assumptions are returned to the user.

The remaining sections of this chapter proceed as follows. Section 7.2 and Section 7.3 present background material and the overall assumption debugging flow, respectively. Section 7.4 describes the method to generate multiple counter-examples. The model to generate fixed cycle assumptions is presented in Section 7.5, and Section 7.6 contains the experimental results. Finally, Section 7.7 concludes this work.

7.2 Preliminaries

7.2.1 Minimal Correction Sets and Unsatisfiable Cores

Recall from Section 2.4.2, given an UNSAT Boolean formula $\Psi$ in CNF, an UNSAT core is a subset of clauses that are unsatisfiable. A Minimal Unsatisfiable Subset (MUS) is an UNSAT core where every proper subset is satisfiable (SAT). A Minimal Correction Set (MCS) is a minimal subset of clauses of $\Psi$ such that removing the subset will result in $\Psi$ being satisfiable. There exists a duality relationship between MUSs and MCSs as it is possible to compute the set of one from the other [88]. Using this relationship, one can calculate all MUSs from all MCSs.

Example 26 The following is an UNSAT Boolean formula written in CNF.

$$\Psi = (a)(\bar{a} + b)(\bar{b})(\bar{a} + c)(c) \quad (7.1)$$

The MCSs of $\Psi$ are: $\{(a)\}, \{(\bar{a} + b)\}, \{\bar{b}\}$. These sets of clauses are MCSs because if any one of these sets of clauses are removed from $\Psi$ then the instance will be satisfiable, and each one of these sets is minimal.

Given an UNSAT CNF formula $\Psi$, MCSs can be computed by introducing a fresh variable to each clause called a relaxation variable. If the variable is active, then the clause is effectively removed from the problem. Using this idea, cardinality constraints [54] can be used to find
all minimal sets of relaxation variables that make $\Psi$ SAT. For each solution, the set of active relaxation variables correspond to a MCS. This idea has been used extensively in modern Max-SAT solvers [90, 97] to compute MCSs.

With respect to debugging, a MUS intuitively represents one way in which a counter-example can excite an error, traverse its effects through the design components and cause a failure at the observation points. In this view, clauses correspond to the counter-example, components of the design and target property. Alternatively, a MCS represents a minimal set of clauses related to components that are potentially erroneous. In other words, removing the components related to the MCS clauses is a potential way to “correct” the design. UNSAT cores and MCSs have been widely used in various debugging applications such as [83, 119, 131].

7.2.2 Related Work

Recent work [9, 109] has studied finding a diverse set of satisfying assignments for a given SAT instance. It uses low-level modifications to the SAT-solver to bias solutions to be as diverse as possible according to a set of defined metrics. This technique is applied in the context of semi-formal verification in order to find corner case bugs that pure formal engines cannot find due to capacity issues. Although related to generating multiple counter-examples, the diversity of the multiple satisfying assignments is based on a heuristic metric that does not necessarily correspond to benefits in the context of debugging.

The situation of generating assumptions has also appeared in other contexts. During constrained random simulation, the work in [143] automatically generates constraint properties to bias the stimulus generator towards missing coverage holes. In compositional verification [35], a key step is generating assumption properties in order to verify the correctness of components in isolation. Previous work [32, 38, 65] aims to automatically generate an assumption on the interface between two components with the goal of proving the target property. Assertion generation techniques [12, 92] have aimed to mine formal properties from simulation traces using formal methods and machine learning techniques. Additionally, generating environmental constraints for software model checking [72, 77] and reactive system synthesis [29, 87] have also been studied. In these situations, the techniques effectively generate properties to accomplish
Chapter 7. Automated Debugging of Missing Assumptions

131

their respective goals. However, none of them addresses the widespread pain of debugging missing input constraints in a formal hardware verification flow.

7.3 Assumption Debugging Flow

This section presents an overview of our methodology for debugging missing assumptions in a formal property checking environment. Although the overall flow is presented in the context of debugging missing input assumptions, as noted earlier, the work here can still be valuable in debugging other types of formal failures such as RTL design errors or incorrectly written assertions. This is because a key by-product of our flow, that is, generating multiple formal counter-examples, can be utilized by “traditional” automated techniques to aid debugging for all types of failures.

The overall methodology is shown in Figure 7.1 and consists of two major phases. Given an assertion failure and its associated counter-example, the first phase attempts to iteratively generate multiple formal counter-examples. For each counter-example, starting with the one returned by the formal tool, a function $F_i$ is extracted that intuitively represents all the failing input behaviors of that counter-example. These functions are then passed back to the formal tool as constraints and used to iteratively generate more counter-examples. This process is repeated until the formal tool cannot return any more counter-examples, or a desired number of counter-examples has been reached.

The second phase of the methodology iteratively generates input assumptions that can prevent failures similar to those seen in the formal counter-examples. This is accomplished by using a model of simple assumption structures and filtering them based on the functions generated in the first phase. The resulting filtered assumptions are returned to the user and can either be used as suggestions for the missing assumptions, or as hints to which signals and expressions might be needed.

This flow improves debugging efficiency in two ways. First, multiple counter-examples can greatly improve debugging of formal failures regardless of their source. This is because they provide a more general representation of the assertion failure, which benefits both manual and
automated debugging techniques [55, 80, 128]. In particular, they can greatly reduce the number of assumptions returned to the user, dramatically improving the quality of results. Second, the assumptions returned by the methodology are easy-to-understand properties based upon “common” assumption structures. This is shown to give better insight into debugging missing assumptions, a fact that is qualitatively examined in experiments. The following sections describe the two phases of our methodology in greater detail.

### 7.4 Generating Multiple Counter-examples

In a typical formal flow when an assertion fails, the formal tool generates a single counter-example to be used later in debugging. Multiple counter-examples are beneficial for debugging because they allow a broader view of the root-cause of failure and may improve the resolution of automated debugging techniques [80, 128]. Despite the benefits of multiple counter-examples, to the best of the author’s knowledge, existing industrial formal property checkers do not support
The difficulty in this process is not simply generating a second counter-example, but rather generating a useful second counter-example that causes the assertion to fail in a different manner. The following sub-sections describe a method to generate multiple formal counter-examples that are quantitatively different from each other. It also outlines how to apply them to filter candidate input assumptions and improve quality of the final results.

### 7.4.1 Minimal Unsatisfiable Input Sets and Minimal Correction Input Sets

This section defines and explains the relationship of two new terms that will be used throughout this chapter.

For a sequential circuit $C$, consider the time-frame expanded circuit modeled with a counter-example of length $k + 1$ and property $P$ in the following equation:\(^2\)

$$\Psi = S^0 \land P \land \left( \bigwedge_{i=0}^{k} X^i \land T^i \right) \quad (7.2)$$

Since $\Psi$ models the counter-example of the unrolled circuit, it is guaranteed to be UNSAT because the counter-example shows how the circuit will cause $P$ to fail. Note that $X^i$ can be modeled as a set of unit clauses since it constrains a single value on each input pin for each time-frame.

From the CNF of Equation \(^2\) we will define two terms relating to its inputs clauses.

**Definition 13** A Minimal Correction Input Set (MCIS), denoted by $C^k$ for the $k^{th}$ such set, is a minimal set of input unit clauses ($C^k \subseteq \bigwedge_{i=0}^{k} X^i$) that when removed, will result in Equation \(^7.2\) being SAT.

This concept is analogous to the idea of MCSs except with respect to only input unit clauses. Thus, $C^k$ can be thought of as a minimal set of input clauses one needs to remove to correct the failure.

**Definition 14** A Minimal Unsatisfiable Input Subset (MUIS), denoted by $U^k$ for the $k^{th}$ such set, is a minimal set of input unit clauses ($U^k \subseteq \bigwedge_{i=0}^{k} X^i$) such that $S^0 \land P \land U^k \land \left( \bigwedge_{i=0}^{k} T^i \right)$

---

\(^2\)Recall, Section \(2.2.1\) defines the symbols for the following Equation.
This concept is analogous to the idea of MUSs except with respect to input unit clauses. Thus, $U^k$ is the minimal set of input unit clauses needed to expose the failure.

From the definition of MUIS, consider the function which represents the disjunction of all MUISs for a given counter-example:

$$F = U^0 + ... + U^k$$

(7.3)

This function intuitively represents all combinations of inputs from the counter-example that can lead to the assertion failure. $F$ can be thought of as the “reasons” for why the counter-example excites the circuit and causes a failure.

Let the $i^{th}$ literal in $U^k$ be denoted by $u^k_i$. We can expand Equation (7.3) to give:

$$F = u^0_0 u^0_1 ... u^{|U^0|}_0 + ... + u^k_0 u^k_1 ... u^{|U^k|}_0$$

$$= (\overline{u^0_0} + \overline{u^0_1} + ... + \overline{u^{|U^0|}_0}) ... (\overline{u^k_0} + \overline{u^k_1} + ... + \overline{u^{|U^k|}_0})$$

(7.4)

Notice that when $F$ evaluates to false, at least one literal in each $U^k$ term is false. In other words, all $U^k$ MUISs can be broken by negating at least one literal from each term in $F$. When this situation occurs, the unrolled counter-example ($\Psi$) will be SAT because there are no longer any MUISs, thus no combination of the modified input stimulus will cause the property to fail. Further, instead of negating these clauses, they can equivalently be removed from the original problem. This situation exactly defines the concept of MCIS. In fact, the relationship between MUISs and MCISs is analogous to the relationship between MUSs and MCSs.

Using this relationship and the fact that these sets only contain unit clauses, $F$ can be simplified and re-written in terms of MCISs ($C^0, \cdots, C^k$). Let the $i^{th}$ literal in $C^k$ be denoted by $c^k_i$. Equation (7.4) can be simplified by distributing the conjunctions over the $u^k_j$ literals and removing redundant terms/literals. Each left over term will correspond to a product of literals.
Thus, $F$ can be computed in a similar manner to computing all MCSs. Begin by adding a fresh relaxation variable to each clause in $\bigwedge_{i=0}^{k} X^i$. Using cardinality constraints, find all minimal SAT solutions with respect to these relaxation variables similar to the process used by modern Max-SAT solvers [90, 97]. Each such solution will correspond to a $C^k$.

Although computing MCSs can be computationally intensive in general, the proposed method only calculates them with respect to the input unit clauses. This allows the method to be more efficient compared to just computing all MCSs.

**Example 27** Consider the implementation of the simple state machine shown in Figure 7.2 that implements a modulo-2 counter that counts up when $a = 1$ and resets if $b = 1$. The property to be verified is:

$$P: s == 2'b01 & & a \implies s == 2'b10$$

Informally, if the counter is at 01 and $a$ is high, then in the next cycle it should be at 10. If sent to a formal property checker, the property will fail because the property was written under the assumption that the reset signal $b$ does not go high. A two cycle counter-example to this property is $X = \langle (a^0, b^0), (a^1, b^1) \rangle$, where the superscripts indicate the clock-cycle. Solving for all MCISs $C^k$, we find: $C^0 = \{a^0\}, C^1 = \{b^0\}, C^2 = \{a^1\}, C^3 = \{b^1\}$. Which can directly be used to build $F = a^0 + b^0 + a^1 + b^1$.

### 7.4.2 Minimal Correction Input Subsets as Blocking Constraints

In the context of debugging, a failure can be viewed as a counter-example exciting an error, propagating its effect through design components, and causing an assertion to fail. This corresponds to the unrolled (in time) CNF of the counter-example from Equation 7.2. The initial

---

Similar to the previous equation, $C^k$ represents a conjunction of all the unit clauses within the $C^k$ set.
Chapter 7. Automated Debugging of Missing Assumptions

states and input vector propagate through the clauses that model the design, and cause a conflict with the modeled property. The corresponding clauses can be abstractly viewed as a set of MUSs. As such, a natural way to quantify two counter-examples as being different is when the observed failures occur with no identical MUSs. This leads to the following definition of distinct counter-examples:

**Definition 15** Given two counter-examples $R$ and $S$, and their respective unrolled CNF instances from Equation 7.2, $\Psi_R$ and $\Psi_S$, let $M_R$ and $M_S$ represent the set of all MUSs from $\Psi_R$ and $\Psi_S$, respectively. Counter-examples $R$ and $S$ are said to be distinct iff $M_r \cap M_s = \emptyset$.

Using this definition, we can generate multiple distinct counter-examples by preventing previously seen MUSs from occurring again. To prevent a MUS, we need to ensure at least one of its clauses is not present. Since the circuit behavior should not change, only the clauses corresponding to the primary input vector should be blocked to prevent previously found MUSs. This corresponds directly to generating a blocking constraint on the inputs to prevent previously found MUSs. Using the duality between MUISs and MCISs, this constraint can be computed from a single MCIS.
Chapter 7. Automated Debugging of Missing Assumptions

Specifically, for the unrolled counter-example $\Psi$ from Equation 7.2 and MCIS $C^k = \{c_0^k, ..., c_{|C^k|}^k\}$, removing $C^k$ will break all MUISs (and thus all MUSs) in $\Psi$ since their removal will make the instance SAT. Since $C^k$ is minimal, this is equivalent to reversing the polarity of the corresponding input unit clauses in $\Psi$, and can be expressed as the following blocking constraint $B^k$ for the $k^{th}$ MCIS:

\[ B^k = \overline{c_0^k} \cdot \overline{c_1^k} \cdot ... \cdot \overline{c_{|C^k|}^k} \]  

(7.6)

This blocking constraint can be used in conjunction with the design and assertion to generate another distinct counter-example using an additional call to the formal tool. The following lemma describes this idea:

Lemma 5 For counter-example $R$, let $\Psi_R$ be the unrolled counter-example in CNF. If $B^k$ is the $k^{th}$ blocking constraint of $\Psi_R$, then any counter-example that satisfies $B^k$ is distinct from $R$.

Proof: From Equation 7.6 $B^k$ is the conjunction of all the negations of literals in the $k^{th}$ MCIS, $C^k$. By definition, removing the literals of $C^k$ from the $\Psi_R$ will result in the instance being SAT, effectively breaking all the MUSs from $\Psi_R$. Since $C^k$ is minimal and no proper subset has the property of being a correction set, any SAT assignment will contain the negation of all the literals of $C^k$, precisely the expression $B^k$. It follows then that any counter-example that contains the assignment from $B^k$ will necessarily not contain any MUSs from $\Psi_R$, and therefore is distinct.

As such, to generate a new distinct counter-example we can use Lemma 5 and pass a blocking constraint in the form of Equation 7.6 to the formal tool. If the formal tool returns a counter-example, it implicitly guarantees that $B^k$ is satisfied, resulting in a distinct counter-example.

7.4.3 A Practical Algorithm

Algorithm 7.1 shows the pseudo-code for generating multiple counter-examples. The algorithm begins by generating the first counter-example from the formal property checker and extracting all MCISs from it (lines 2-5). The loop from line 6 generates multiple counter-examples. For
a given MCIS, it will attempt to find a new counter-example. If successful (line 9), this MCIS is saved in blocking to ensure that future counter-examples remain distinct. This process greedily selects new MCISs to add to blocking only when it can find a new counter-example. Once the new counter-example is saved, a new set of MCISs is extracted (lines 11-12), and the process repeats using this new set of MCISs. The loop stops when either none of MCISs combined with the existing constraints in blocking can generate another counter-example, or when the maximum number of user-specified counter-examples has been reached. The following theorem confirms the benefits of these counter-examples:

**Theorem 4** All counter-examples returned by Algorithm 7.1 are mutually distinct.

*Proof:* Each counter-example generated from the run of the formal tool on line 8 will run under the set of blocking constraints, blocking $\cup$ C. By Lemma 5, any counter-examples that derived any of the constraints in blocking $\cup$ C will be distinct from the newly generated one. Since blocking constraints are added only when a new counter-example is found, blocking $\cup$ C maintains a set of MCISs from each of the previously seen counter-examples. Therefore, each newly generated counter-example will respect these blocking constraints and will be mutually distinct.

One important aspect of Algorithm 7.1 is that it iteratively adds a blocking constraint in the form of Equation 7.6. This could have alternatively been implemented using the disjunction of all blocking clauses from a single counter-example *i.e.*, the negation of Equation 7.5. However, our experience with an industrial formal property checker shows that this latter approach significantly slows down the tool causing time-outs or bounded proofs. This observation can be explained as follows. Our blocking constraints are just unit clauses, which are easily modeled within many different model checking algorithms. Whereas, the disjunction of multiple MCISs can be significantly more complicated to model (or at least require specialized optimizations). This allows for a more generic method without any need to use a specialized property checker.
Algorithm 7.1 Generating Multiple Counter-Examples

1: procedure MULTIPLECOUNTEREXAMPLES(max)
2:     blocking = ∅
3:     c-ex = RUNFORMAL(blocking)
4:     CEX = {c-ex}
5:     MCIS = EXTRACTALLMCIS(c-ex)
6:     while MCIS ≠ ∅ and | CEX | < max do
7:         C = EXTRACTBLOCKING(MCIS)
8:         c-ex = RUNFORMAL(blocking ∪ C)
9:         if c-ex ≠ ∅ then
10:             blocking = blocking ∪ C
11:             CEX = CEX ∪ c-ex
12:             MCIS = EXTRACTALLMCIS(c-ex)
13:         end if
14:     end while
15: return CEX
16: end procedure

7.4.4 Applications for Debugging Missing Input Assumptions

Recall the disjunction of all MUISs for a given counter-example, \( F \), from Equation 7.3. This function represents all combinations of inputs from the counter-example that can lead to the assertion failure. Given a candidate assumption, \( A \), we can use \( F \) to determine whether \( A \) will prevent the failing input behaviors seen in \( F \).

In particular, if \( F \cdot A \) is SAT, then \( A \) does not prevent the failure given in the counter-example since at least one of MUIS \( (U^k) \) is SAT. Inversely, if \( F \cdot A \) is UNSAT, then \( A \) will ensure that future failures will not occur in the same way as the given counter-example. However in the latter case, \( A \) may not constrain the input space enough to prevent all failures, but it at least prevents failures similar to those seen in the counter-example. This concept can similarly be extended to multiple counter-examples.
If Algorithm 7.1 is used to derive multiple counter-examples, all MCISs from each counter-example are indirectly generated as a by-product. From Equation 7.5, these can be used to generate a set of filtering functions $F_1, ..., F_N$ for $N$ counter-examples, respectively, which can naturally be combined to extend the filtering function and generate the following instance:

$$\psi = (F_1 + ... + F_N) \cdot A \tag{7.7}$$

The disjunction of all the $F_i$ in Equation 7.7 correspond to all the MUISs for each of the counter-examples. This implicitly encodes all the input behaviors that led to the observed assertion failures in the given counter-examples. If the instance is SAT, then the assumption is not strong enough to prevent at least one of the observed failures. Otherwise, the assumption is generalized enough to prevent all the observed failures and should be returned to the user.

It should be noted that although we present this filtering function in the context of pruning generated assumptions in this Chapter, it is equally valid to say that this function can be used to test manually generated assumptions by the engineer. Thus, the filtering function can provide quick feedback to determine if a given assumption can prevent the failure(s) present in the current counter-example(s).

**Example 28** Consider the filtering function $F$ generated from Example 27 and the four stuck-at fault properties that would be generated: $a, \overline{a}, b,$ and $\overline{b}$. Of these, only the first one would be filtered out since it would return SAT when run with $F$, while the others all return UNSAT. Of the remaining, it is easy to see how they translate to high-level behavior of the design: $a$ prevents the counter from incrementing (a vacuous condition), $b$ continually resets the machine (also vacuous), and $\overline{b}$ turns off reset (desired result).

### 7.5 Generating Assumptions

As mentioned in Section 7.4.4, a user can check if their manually written assumption is valid using the techniques from the previous section. However, it is often helpful to give additional information to help debug missing assumptions, allowing better intuition and insight into which signals, expressions and properties are required. Thus, the *goal* of automatically generating
assumptions is to supplement the user’s expert design knowledge with hints or suggestions so that the missing assumption can be more quickly determined.

### 7.5.1 Assumption Model

Table 7.1 shows a summary of the model used to generate candidate missing input assumptions. Each row corresponds to one of four categories of properties presented in SystemVerilog. These categories correspond to simple unit Booleans, combined Boolean operators, one-hot operators, and stability expressions. An assumption is generated by taking the property and using the same clock and reset as the target failing assertion. Each assumption is then checked against the filtering function to determine if it should be returned to the user.

In the table, \texttt{input} refers to a single bit primary input pin, while \texttt{bus} refers to a semantic grouping of primary input pins. While these two types of signals can be extracted directly from the RTL description, in some cases user-input can greatly enhance the strength of the model. For instance, in certain cases, a Verilog bus grouping may logically correspond to multiple smaller bits or buses, and should be broken down accordingly. This provides a better context for the user when analyzing the generated assumptions.

The assumptions are purposefully simple, fixed-cycle properties to provide effective explanatory power while debugging. The motivation behind this decision was that any algorithmically generated property is unlikely to be helpful because such properties will be both difficult to understand and potentially expensive to compute. This results in a flow that is impractical for a user who must both understand the context of the assumption, as well as be able to debug it efficiently.

In contrast, the simplicity of the properties provides easy to understand feedback to the users, giving them hints to the actual missing assumption, and in some simple cases generating the actual missing assumption. A detailed analysis of the qualitative benefits are shown in the experimental results.
Table 7.1: Assumption Model

<table>
<thead>
<tr>
<th>Category</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit Booleans (unit)</td>
<td>input, !input</td>
</tr>
<tr>
<td>Combined Booleans</td>
<td>&lt;unit&gt; &amp; &lt;unit&gt;, &lt;unit&gt; &amp; &lt;unit&gt; &amp; &lt;unit&gt;, &lt;unit&gt;</td>
</tr>
<tr>
<td>One-hot</td>
<td>$onehot(bus), $onehot0(bus), $onehot({&lt;unit&gt;, &lt;unit}&gt;), $onehot({&lt;unit&gt;, &lt;unit&gt;, &lt;unit&gt;})</td>
</tr>
<tr>
<td>Stability</td>
<td>$stable(bus), bus == 0 input</td>
</tr>
</tbody>
</table>

7.5.2 Pruning Candidate Assumptions

As can be seen from Table 7.1, several of the properties depend on combining smaller unit Boolean expressions to create more complex ones. This can cause a significant amount of properties to be generated that are not necessarily useful.

To deal with this issue, we iteratively filter the candidate input assumptions starting with the unit Booleans properties. If any unit Boolean is shown to pass the filter, it is returned to the user and not considered for any other categories. Otherwise, the unit Boolean will be considered in more complex properties. This has the potential to dramatically reduce the number of generated assumptions due to the multiplicative nature of the more complex properties.

The rationale behind this approach is that a single unit Boolean property provides the simplest way to convey the importance of this signal to the user. While it is possible to gather additional information by using it in more complex properties, it adds a significant amount of noise to the generated results. This translates into many structurally similar generated assumptions that the user must manually examine, partially negating the benefit of automated generated results. This is an important consideration since too many generated assumptions significantly reduce the benefit of this technique, resulting in longer debug times to filter through the results.
7.6 Experimental Results

This section presents experimental results for the proposed methodology. All experiments are performed on a single core of an Intel Core i5 3.1 GHz quad-core workstation with 16 GB of RAM. A commercial property checker [24] is used with default settings to perform all formal checks, while the extraction of MCISs as well as generation and filtering of candidate assumptions are all implemented in C++, using Minisat [53] as the SAT engine. A commercial SystemVerilog frontend [139] is used to parse and elaborate all designs, assertions, and assumptions in this chapter. The designs in this chapter may be used in experiments for other chapters, but may differ in their design characteristics. This difference can attributed to the version of the design and the version of the frontend.

Nine designs are selected for evaluation from OpenCores [111] with assertions written based upon their specification documents. These assertions consisted of ones that came with the OpenCores design, and ones written by us from the accompanying specification documents. Table 7.2 presents relevant statistics and a description for each design used in the experimental results. The columns of the table list the design name, number of gates (including state elements), number of state elements, number of primary input pins, and a brief description of the design.

For each gathered design and assertion, the formal property checker is run and any failure is considered to be an instance of a missing assumption. The instances listed in the following tables correspond to a single failing assertion and are labeled by appending a number to the design name.

7.6.1 Generating Multiple Counter-Examples

This section presents experimental results for the proposed approach to generate multiple counter-examples from Section 7.4. Experiments in this subsection are conducted for each instance by, first, running Algorithm 7.1 to generate as many counter-examples as possible within 1800 seconds to a maximum of 15. Next, using either 1, 5, 10, or 15 counter-examples, candidate assumptions are generated and filtered to examine if multiple counter-examples are
useful in reducing the number of generated assumptions. To ensure that the same number of assumptions are used for each case, combined and one-hot type properties from Table 7.1 are omitted since these depend on pruning. Additionally, each pin of an input bus is also used in unit Boolean properties so that we have a sufficient set of properties across all input pins. Note that a cone of influence [35] optimization is run on the failing assertion of each instance, resulting in a potentially different number of total generated assumptions between instances of the same design. Table 7.3 shows the results of these experiments.

The first five columns list the instance name, number of counter-examples generated, run-time to extract MCISs from the counter-examples, run-time of the formal tool to generate that many counter-examples, and the total number of candidate assumptions for that instance. The last four columns show how many of the candidate assumptions remain after filtering using the technique from Section 7.4.4 with 1, 5, 10, and 15 counter-examples, respectively. A “-” in the last four columns means that many counter-examples could not be generated within the time limit.

Overall the last four columns show that using more counter-examples can effectively reduce the number of filtered assumptions. On average, for instances that are able to generate either 5, 10, or 15 counter-examples, the number of filtered assumptions are reduced by 30.4%, 37.9% and 38.3%, respectively, compared to a single counter-example. This confirms that the additional counter-examples generated do generalize the assertion failure, which shows that our definition of distinct is indeed a valid one.
Chapter 7. Automated Debugging of Missing Assumptions

The ability of the proposed technique to filter candidate assumptions works well in most of the instances (such as \texttt{ddr2.2} and \texttt{usbfunct.1}), but not all (such as \texttt{cpu.1} and \texttt{mips.2}). This can be explained as follows. The former case is the ideal behavior where the second counter-example does indeed find a different way to excite the design and cause the assertion to fail. While the latter case finds a counter-example similar to the original one but shifted in time. One needs to note that in this situation, it is often the case that there may only be one way to cause the assertion fail.

When analyzing the run-time, there are two main contributors. The first is the extraction of MCISs, which depends on the size of the design, number of input pins, and the length of the counter-example. For many cases, such as \texttt{risc.1} and \texttt{wb.1}, it is relatively fast. In the case of \texttt{ddr2.1}, however, the excessive number of inputs (431) cause the run-time of extracting the MCISs to be large. The other contributor to overall run-time is multiple iterations of the loop in Algorithm 7.1, which may require many calls to the formal tool before a counter-example is found. However, within the 1800 second timeout, 7 out of the 18 instances were able to generate 15 counter-examples, and 16 out of the 18 were able to generate at least 5. This shows that this technique is effective in generating multiple counter-examples within a short amount of time.

### 7.6.2 Assumption Debugging Methodology

This section presents experimental results for the overall assumption debugging methodology from Section 7.3. For each instance, counter-examples are generated within a time limit of 1800 seconds up to a maximum of 10. Additionally, the full assumption model from Section 7.5 is used to generate and filter assumptions. Note that this may result in a different number of candidate assumptions compared to the previous subsection. Similarly, a cone of influence optimization is run on the failing assertion for each instance.

Table 7.4 shows the quantitative results of these experiments. The qualitative results are detailed in Section 7.6.3. The columns list the instance name, number of counter-examples generated, time to extract MCISs from the counter-examples, time of the formal tool to generate that many counter-examples, time to generate and filter candidate assumptions, total number of candidate assumptions, and the number of assumptions after filtering.
Table 7.3: Experiments showing the effect of using multiple counter-examples to filter assumption candidates.

<table>
<thead>
<tr>
<th>Instance Name</th>
<th># of Counter-Examples</th>
<th>MCIS Formal Time (s)</th>
<th>Total Candidates</th>
<th># of Candidates Remaining After Filtering with n Counter-Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu_1</td>
<td>15</td>
<td>653</td>
<td>356</td>
<td>154</td>
</tr>
<tr>
<td>cpu_2</td>
<td>10</td>
<td>778</td>
<td>616</td>
<td>154</td>
</tr>
<tr>
<td>ddr2_1</td>
<td>3</td>
<td>625</td>
<td>86</td>
<td>226</td>
</tr>
<tr>
<td>ddr2_2</td>
<td>9</td>
<td>383</td>
<td>1395</td>
<td>257</td>
</tr>
<tr>
<td>hpdmc_1</td>
<td>15</td>
<td>112</td>
<td>148</td>
<td>97</td>
</tr>
<tr>
<td>hpdmc_2</td>
<td>15</td>
<td>123</td>
<td>200</td>
<td>97</td>
</tr>
<tr>
<td>mips_1</td>
<td>4</td>
<td>278</td>
<td>93</td>
<td>163</td>
</tr>
<tr>
<td>mips_2</td>
<td>12</td>
<td>813</td>
<td>959</td>
<td>163</td>
</tr>
<tr>
<td>risc_1</td>
<td>8</td>
<td>88</td>
<td>1126</td>
<td>92</td>
</tr>
<tr>
<td>risc_2</td>
<td>7</td>
<td>190</td>
<td>1339</td>
<td>92</td>
</tr>
<tr>
<td>pci_1</td>
<td>8</td>
<td>611</td>
<td>761</td>
<td>267</td>
</tr>
<tr>
<td>pci_2</td>
<td>8</td>
<td>648</td>
<td>723</td>
<td>267</td>
</tr>
<tr>
<td>spi_1</td>
<td>15</td>
<td>8</td>
<td>177</td>
<td>22</td>
</tr>
<tr>
<td>spi_2</td>
<td>15</td>
<td>47</td>
<td>214</td>
<td>22</td>
</tr>
<tr>
<td>usbfunct_1</td>
<td>12</td>
<td>901</td>
<td>858</td>
<td>131</td>
</tr>
<tr>
<td>usbfunct_2</td>
<td>10</td>
<td>186</td>
<td>1152</td>
<td>131</td>
</tr>
<tr>
<td>wb_1</td>
<td>15</td>
<td>6</td>
<td>846</td>
<td>13</td>
</tr>
<tr>
<td>wb_2</td>
<td>15</td>
<td>8</td>
<td>344</td>
<td>41</td>
</tr>
</tbody>
</table>

From column 7, the absolute number of filtered assumptions returned to the user is relatively small with an average of 28. It is important that this number is not too large, or else the list of assumptions may become unwieldy for a user to analyze. Although most of the instances fall close to this average, there is one outlier ddr2_2 with 333 returned assumptions. This is due to the large number of input pins which generates a significant number of candidate assumptions (4094). However, as described in Section 7.6.3, the different categories of properties allow one to narrow down the analysis. In this case, only analyzing the unit Booleans assumptions proved most useful.

The benefit of the pruning technique from Section 7.5.2 can be seen in the total number of candidates for usbfunct_1 and usbfunct_2, which are 148 and 1135, respectively. In the former case, usbfunct_1 returned several unit Booleans which significantly reduced the number of candidate assumptions in more complex properties, while usbfunct_2 did not return any unit Booleans. Most of the other instances behaved like the former case allowing the pruning
Table 7.4: Experimental results for the overall assumption debugging methodology from Section 7.3.

<table>
<thead>
<tr>
<th>Instance Name</th>
<th># of Counter-Examples</th>
<th>MCIS Extraction Time (s)</th>
<th>Formal Time (s)</th>
<th>Assumption Generation and Filtering Time (s)</th>
<th>Total # of Candidates</th>
<th># of Candidates After Filtering</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu_1</td>
<td>10</td>
<td>255</td>
<td>100</td>
<td>5</td>
<td>31</td>
<td>3</td>
</tr>
<tr>
<td>cpu_2</td>
<td>10</td>
<td>778</td>
<td>616</td>
<td>7</td>
<td>28</td>
<td>5</td>
</tr>
<tr>
<td>ddr2_1</td>
<td>3</td>
<td>625</td>
<td>86</td>
<td>TO</td>
<td>857</td>
<td>21</td>
</tr>
<tr>
<td>ddr2_2</td>
<td>9</td>
<td>383</td>
<td>1395</td>
<td>1504</td>
<td>4094</td>
<td>333</td>
</tr>
<tr>
<td>hpdmc_1</td>
<td>10</td>
<td>70</td>
<td>60</td>
<td>10</td>
<td>90</td>
<td>33</td>
</tr>
<tr>
<td>hpdmc_2</td>
<td>10</td>
<td>77</td>
<td>65</td>
<td>8</td>
<td>65</td>
<td>18</td>
</tr>
<tr>
<td>hpdmc_3</td>
<td>10</td>
<td>6</td>
<td>77</td>
<td>1</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>mips_1</td>
<td>4</td>
<td>278</td>
<td>93</td>
<td>9</td>
<td>59</td>
<td>22</td>
</tr>
<tr>
<td>mips_2</td>
<td>10</td>
<td>455</td>
<td>276</td>
<td>8</td>
<td>39</td>
<td>10</td>
</tr>
<tr>
<td>mips_3</td>
<td>5</td>
<td>134</td>
<td>458</td>
<td>7</td>
<td>39</td>
<td>6</td>
</tr>
<tr>
<td>mips_4</td>
<td>10</td>
<td>589</td>
<td>631</td>
<td>10</td>
<td>59</td>
<td>7</td>
</tr>
<tr>
<td>risc_1</td>
<td>8</td>
<td>88</td>
<td>1126</td>
<td>5</td>
<td>39</td>
<td>10</td>
</tr>
<tr>
<td>risc_2</td>
<td>7</td>
<td>190</td>
<td>1339</td>
<td>6</td>
<td>34</td>
<td>9</td>
</tr>
<tr>
<td>risc_3</td>
<td>5</td>
<td>79</td>
<td>169</td>
<td>4</td>
<td>20</td>
<td>9</td>
</tr>
<tr>
<td>risc_4</td>
<td>9</td>
<td>116</td>
<td>898</td>
<td>5</td>
<td>39</td>
<td>14</td>
</tr>
<tr>
<td>pci_1</td>
<td>8</td>
<td>611</td>
<td>761</td>
<td>7</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>pci_2</td>
<td>8</td>
<td>648</td>
<td>723</td>
<td>7</td>
<td>22</td>
<td>11</td>
</tr>
<tr>
<td>pci_3</td>
<td>8</td>
<td>564</td>
<td>518</td>
<td>8</td>
<td>25</td>
<td>10</td>
</tr>
<tr>
<td>pci_4</td>
<td>2</td>
<td>466</td>
<td>60</td>
<td>27</td>
<td>261</td>
<td>82</td>
</tr>
<tr>
<td>spi_1</td>
<td>10</td>
<td>4</td>
<td>48</td>
<td>1</td>
<td>20</td>
<td>9</td>
</tr>
<tr>
<td>spi_2</td>
<td>10</td>
<td>28</td>
<td>60</td>
<td>15</td>
<td>74</td>
<td>29</td>
</tr>
<tr>
<td>usbfunct_1</td>
<td>10</td>
<td>737</td>
<td>334</td>
<td>14</td>
<td>148</td>
<td>58</td>
</tr>
<tr>
<td>usbfunct_2</td>
<td>10</td>
<td>186</td>
<td>1152</td>
<td>132</td>
<td>1135</td>
<td>44</td>
</tr>
<tr>
<td>usbfunct_3</td>
<td>10</td>
<td>18</td>
<td>244</td>
<td>2</td>
<td>16</td>
<td>7</td>
</tr>
<tr>
<td>wb_1</td>
<td>10</td>
<td>3</td>
<td>123</td>
<td>1</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>wb_2</td>
<td>10</td>
<td>4</td>
<td>111</td>
<td>1</td>
<td>81</td>
<td>2</td>
</tr>
<tr>
<td>wb_3</td>
<td>10</td>
<td>5</td>
<td>79</td>
<td>1</td>
<td>19</td>
<td>2</td>
</tr>
<tr>
<td>wb_4</td>
<td>10</td>
<td>4</td>
<td>98</td>
<td>1</td>
<td>81</td>
<td>2</td>
</tr>
</tbody>
</table>
technique to successfully reduce the number of candidate assumptions.

When analyzing run-time of generating and filtering candidates in column 5, in most instances the time is relatively small and both tasks can be completed within 60 seconds. However, ddr_1 and ddr_2 are again outliers, where the former hit a time limit of 1800 seconds. Here, the excessive number of input pins cause an exponential number of generated assumptions in the more complex properties. In these cases, it may be more prudent to only generate simpler properties or limit the number of pins used to generate assumptions. Both these solutions are easily implementable within the proposed flow.

7.6.3 Qualitative Analysis of Generated Assumptions

The following is a detailed discussion on the qualitative aspects of how the proposed flow can be used to aid debugging of missing assumptions. We selected seven instances from Table 7.4 to illustrate the benefits.

The qualitative results are derived by working with a verification engineer\(^5\) to analyze and comment on the results of the proposed flow. The verification engineer’s overall impression of the methodology is that it is a useful first step in debugging the missing assumption. Although unlikely to generate the exact assumption needed, the generated assumptions instead remind the engineer of relevant signals and expressions, speeding up the initial work of debugging by obtaining the context of the failure more quickly.

**cpu_1:** The assertion for this instance is for the decoder unit of the processor. It checks that the INSTR_SEQ register is set correctly:

\[
P: \text{(INSTR_SEQ != 4'b0000)}
|-> ( \sim \text{past}(\text{SLOT}) \text{|| past(\text{NEXT_IDSTALL}) || past(RST) == 1'b1})
\]

The proposed approach generated 3 constraints, listed here:

\[
A1: \text{!SLOT}
A2: \text{IF_STALL}
\]

\(^5\)Evean Qin, Senior Application Engineer, Vennsa Technologies, Inc.
A3: !IF_STALL & SLOT

The actual bug in this case is inside the assertion’s antecedent (and not a missing assumption) where the following expression is needed: \((\text{INSTR}_\text{SEQ} == \$\text{past} (\text{INSTR}_\text{SEQ}))\). In this case, all three assumptions suggest a useful fix for the assertion since some combination of SLOT and IF_STALL essentially force INSTR_SEQ to be held, effectively the behavior needed in the antecedent.

**ddr2.2:** For this instance, the given assertion is checking: *the minimum time between activate and a READ or WRITE command is \(t_{RCD}\) less the additive latency.*

\[
P: (|t_{rcd\_al\_cnt}) |\rightarrow (\neg \text{read}_i \&\& \neg \text{read\_precharge}_i \\
\&\& \neg \text{write}_i \&\& \neg \text{write\_precharge}_i)\]

The approach generates 333 assumptions, by far the largest number compared to other instances. This is due to the large number of primary input pins (*i.e.*, 431) for this design. However, a large number of these assumptions are combined Booleans and one-hot type properties. Since these assumptions are more complicated and difficult to understand, we focused on assumptions with a single signal, several of which are listed here:

A1: !free
A2: free_cas_cnt_proxy
A3: pt_ch_blk_new_openbank
A4: !l2if_rd_req
A5: ch0_que_channel_picked
A6: other_que_pos[4:0] == 5'b00000
A7: ch0_que_ras_int_picked[7:0] == 8'b00000000

The first two assumptions on free and free_cas_cnt_proxy are on undriven cut-points. Since these cut-points are used to reduce the property checking complexity, they most likely are not good candidates for putting an assumption as they may over constrain the problem.

The other assumptions hint at the need for multiple complex assumptions. For example, ch0_que_ras_int_picked should be always turned off to enable column address selection, ch0_que_channel_picked should be high to select the same channel during the assertion and
$\text{ldif_rd_req}$ can be deasserted to enable the read. Additionally, $\text{other_que_pos}$ should be turned off all the time by picking 0 or another unrecognized address (e.g., address 'ha,'hb, and 'hd are recognizable addresses in this case). Although the required assumptions are complex in this instance, the generated assumptions point the user in the right direction.

**hpdmc.1:** The assertion for this instance specifies that when the read signal is asserted, the ack signal be asserted a certain number of cycles later depending on the $\text{tim\_cas}$ register:

$$P: \text{rose(read)} \implies (!\text{tim\_cas } \#5 \text{rose(ack)})$$

$$\text{or } (\text{tim\_cas } \#6 \text{rose(ack)})$$

The proposed approach generated 33 assumptions, the most useful being these unit Boolean assumptions:

- $A_1$: $\neg\text{wbc\_we\_i}$
- $A_2$: $\neg\text{wbc\_stb\_i}$
- $A_3$: $\neg\text{wbc\_cyc\_i}$
- $A_4$: $\text{fml\_we}$
- $A_5$: $\neg\text{fml\_stb}$

The first three unit assumptions basically translate to not over-writing the $\text{tim\_cas}$ register, while the last two attempt to change the timing of the ack signal. Both of these sets of assumptions point to two types of functionally different behaviors that can be added to ensure that the environmental behavior is modeled properly. Although the missing assumption may not be exactly of this form, it is likely that an engineer who is familiar with the design will be able to write the assumption quickly.

**mips.1:** The assertion for this instance checks to see that a finite state machine transition occurs with the correct conditions:

$$P: \text{CurrState} == \text{"IDLE"} \&\& \text{irq} \&\& \neg\text{iack}$$

$$\implies (\text{CurrState} == \text{"IRQ"})$$

The approach generated 22 assumptions, a sample of which is listed here:
A1: !pause
A2: pause
A3: $onehot(zz_ins_i[31:0])
A4: $onehot0(zz_ins_i[31:0])

In this case, both pause and its negation are suggested by the technique. This is because holding pause either high or low for the entire trace will prevent the assertion from failing. This is a common occurrence in many of the generated assumptions, however, usually only one of the two stuck-at assumptions will be relevant.

By tracing the relationship between pause and the state machine, it is clear that when pause is asserted, the state transition will be stopped. In this case, A1 is precisely the needed constraint. Interestingly, the specification does not explicitly mention that the state transition will be stopped by the pause signal, a common omission that causes counter-examples due to missing assumptions.

Finally, the last two assumptions are on the primary input bus corresponding to the CPU’s instruction. They are obviously not very meaningful and correspond to a vacuous fix.

risc_1: The assertion in this instance is checking the BTFSC instruction which tests a bit of a register if it is 0 and sets an appropriate flag:

P: (inst_data[11:8]==I_BTFSC[11:8]) #=> valid
    |=> ~valid

The approach generated 10 assumptions. Here, inst_data[11:0] was broken down into smaller buses corresponding to the different semantics of each of the bit ranges. Several of the interesting assumptions are listed:

A1: inst_data[4:0] == 5’b00000
A3: inst_data[11:0] == 12’b000000000000
A4: $onehot(inst_data[11:6])
A5: $onehot0(inst_data[11:6])
A6: $onehot(inst_data[11:0])
A7: $onehot0(inst_data[11:0])
The assumptions on `inst_data[11:6]([11:0])` change the instruction the assertion is checking, so they are vacuous fixes. However, the assumption on `inst_data[4:0]` is more interesting. This part of the instruction data controls the register file address. When it is set to 0, it reads from the indirect register, and when it stays the same, it bypasses the register file altogether. Thus, these assumptions give a good hint that when the BTFSC instruction is called, the register file address should remain the same (or must have the same value as the indirect register) until it is finished.

**pci.1:** This assertion in this instance checks the following design intent: Once Target has asserted STOP, it keeps STOP asserted until FRAME is deasserted then stop is deasserted:

```
P: stopn ##1 !stopn & & !framen
  |-> !stopn [*1:$] #0 $rose(framen) #1 $rose(stopn)
```

The approach generated 10 assumptions of which several interesting ones are listed here:

- A1: `!pci_frame_i`
- A2: `pci_frame_i`
- A3: `!pci_idsel_i`
- A4: `pci_irdy_i`

The first assumption specifies that `pci_frame_i` should be held high for the entire trace, a vacuous fix. While the assumption for `pci_frame_i` low will prevent the consequent from completing due to [*1:$], another vacuous fix.

Assumptions on `pci_idsel_i` and `pci_irdy_i` on the other hand, point out that these signals should not be changed during the assertion. They are good hints, which could form the basis for a more complex assumption.

**usbsfunc.1:** The assertion for this instance checks the following property: Buffer overflow occurs when a packet has been received that does not fit into the buffer. The packet will be discarded and a NACK will be sent to the host.

```
P: buffer_overflow #0 send_token[->1]
  |-> (token_pid_sel == NACK)
```
The approach generated 58 assumptions, several of which are listed here:

\begin{align*}
A1: & !wb\_stb\_i \\
A2: & !wb\_cyc\_i \\
A3: & !wb\_addr\_i[17] & wb\_cyc\_i & wb\_stb\_i \\
A4: & !RxValid\_pad\_i \\
A5: & RxError\_pad\_i \\
A6: & !RxActive\_pad\_i \\
A7: & RxActive\_pad\_i \\
A8: & $stable(DataIn\_pad\_i[7:0])
\end{align*}

The first two assumptions that pull down \texttt{wb\_stb\_i} and \texttt{wb\_cyc\_i} are vacuous fixes. But the assumptions with both of these signals high (A3) are more interesting. In this assumption, the 17\textsuperscript{th} bit in \texttt{wb\_addr\_i} controls the source of the data to be sent. This assumption tells the user that during the assertion, the data should be selected from the register file instead of memory, avoiding the cause for the failure.

The next few assumptions on signals \texttt{RxValid\_pad\_i}, \texttt{RxError\_pad\_i} and \texttt{RxActive\_pad\_i} are all vacuous fixes, while the assumption $\texttt{stable(DataIn\_pad\_i[7:0])}$ provides a useful hint. This signal controls which data will be selected from the endpoint. It tells the user that during the assertion, the same endpoint should be selected, providing another way to avoid the failure.

### 7.7 Summary

In this work, a novel debug automation methodology for missing input assumptions is presented. It begins by generating multiple formal counter-examples for the failure along with a function that encodes the input combinations that caused the assertion to fail. This function is later used to generate a list of fixed cycle assumptions that prevent the failures seen in the counter-examples, which can then be used as hints for the actual missing assumption. An extensive set of experimental results on OpenCores designs and assertions show the efficacy of this work.
Chapter 8

Conclusions and Future Work

8.1 Contributions

Modern integrated circuit design has become an immensely complex process with an ever growing list of challenges for the electronic design community. Despite advances in many areas of the design flow, functional debugging still remains one of the key challenges where an overwhelming amount of manual effort is required by the engineer. This manual debug effort has grown in size and scope as bugs commonly appear in both the design and verification environment, exacerbating the existing challenges. As with many other aspects of the design, greater use of automation is necessary to alleviate the burden of increased debugging size and scope.

Over the past decade, research in design debugging based on formal methods \[129\] has shown great promise in providing solutions to the increased size of debugging problems. Techniques to scale \[81, 82, 121, 131\] these methods have shown orders-of-magnitude increase over the basic formulation. Despite these efforts, automated design debugging engines are only practical in a subset of industrial use cases due to the capacity limitations of these techniques. This limits the applicability of these tools when dealing with the ever growing size of design debugging problems.

Meanwhile, a new class of errors that increase the scope of functional debugging have emerged in the verification environment. The massive amounts of verification code that is written is just as susceptible to bugs as design code. Thus, a failure due to a verification en-
environment or design bug still takes significant manual engineering effort. This greatly expands the debugging scope which increases the challenges in an area where automation is still in its infancy.

This dissertation aims to address these two key debugging challenges of increased size and scope. It presents multiple contributions that both scale existing automated design debugging techniques, as well as introduce tools to target two specific aspects of debugging challenges in the verification environment. In summary, this thesis makes the following contributions:

- In Chapter 3, an abstraction and refinement algorithm for design debugging is introduced. This work improves upon past abstraction and refinement schemes by guaranteeing sound and complete results for the overall algorithm. It begins by defining an initial abstraction that removes all components of the design, leaving only the primary inputs/outputs. It then iteratively adds back logic within hierarchical modules guided by the use of UNSAT cores, addressing primarily the design size aspect of debugging complexity. At each iteration, the abstraction is shown to represent a strict under-approximation of the concrete debugging problem, which guarantees sound results. Experimental results show the technique is able to return solutions for 19 of the 22 instances compared to only 9 when the technique is not used. The extension provides further benefits as all 22 instances return solutions at the cost of increased refinement iterations. This contribution is published in [83].

- In Chapter 4, a design debugging algorithm utilizing time-windows in an abstraction and refinement framework is presented. This contribution addresses the error trace length aspect of debugging complexity with its novel use of time-windows. The process begins by dividing the error trace into non-overlapping time-windows where each window is analyzed separately. Similar to other window-based methods, only one time-window is explicitly modeled while subsequent time-frames are over-approximated using the abstraction. The key novelty in this work is the use of a path-based abstraction to model parts of the error trace that directly lead to the observed failure during simulation. This is accomplished by modeling only those paths in the time-frame expanded circuit that directly affect the
observed failure. Additionally, unlike previous work, it provides an efficient method to refine this abstraction using an iterative technique based upon the conflict graph returned by the SAT-solver. The proposed approach can analyze error traces that are 64.6% longer while simultaneously using significantly less memory when compared to previous work. Additionally, the refinement strategy of the flexible algorithm is able to generate only 1.5% additional spurious solutions compared to 114.7% when no refinement is used. This path-based debugging algorithm is published in [85].

- In Chapter 5, an algorithm to both generate and apply unsatisfiable cores for debugging multiple design errors is presented. It improves upon previous work by showing how to generate UNSAT cores without the need to find multiple overlapping UNSAT cores. The generation of the UNSAT cores comes as a by-product of solving the SAT-based debugging formulation at each error cardinality. By using the cores generated at each cardinality, many locations can be ruled out that are provably not part of solutions at higher error cardinalities. This effectively reduces the exponential solution space for multiple simultaneous design errors. Experimental results show that for finding all equivalent errors up to error cardinality three, the core technique is able to reduce the total run-time on average by 22% with a negligible impact on peak memory. This work is published in [78].

- In Chapter 6, an automated debugging methodology for modern assertion languages is presented. It takes a different approach compared to traditional localization-based RTL debug automation methods. Instead of localizing the error, the methodology aids debugging by generating a set of properties closely related to the original failing assertion, where each generated property has been validated with respect to the RTL. These properties serve as a basis for possible corrections to the failing assertion, providing an intuitive method for both correction and debugging. They also provide insight into design behavior by being able to contrast their differences with the failing assertion. This results in increased debugging efficiency by removing much of the manual guesswork needed when debugging assertions. An extensive set of experimental results are presented on real designs with SystemVerilog assertions written from their specifications. The results show
that the proposed methodology and modification model are able to return high quality verified properties for all instances. In addition, the multiple error and vacuity techniques are able to filter out inessential properties by an average of 23% and 34% respectively. This assertion debugging methodology is published in [79, 80].

- In Chapter 7, an algorithm is presented to automatically generate missing input assumptions given a failing counter-example. It begins by generating multiple formal counter-examples for the error. From these counter-examples, a minimal set of failing input behaviors from the counter-example are extracted. Next, from these failing input behaviors, a logic function is constructed that encodes these bad input combinations responsible for the failure. Using this function, small SAT instances are created to filter out a dictionary of fixed cycle properties based on the input signals. The net result returned to the user is a list of input assumptions that restrict the bad behaviors seen in the counter-examples. This list can either contain the actual missing assumption, or may give strong intuition about which assumption is needed, greatly simplifying the tedious process of manually debugging these failures. An extensive set of experimental results have been performed over a wide variety of designs with SystemVerilog assertions written from their specification documents. Multiple counter-examples are shown to reduce the number of generated assumptions by 38% on average for ten counter-examples with an average of 28 assumptions returned to the user. A preliminary version of this work is published in [84].

8.2 Future Work

This dissertation presented several novel contributions to debug automation in a modern verification environment. It introduces several techniques to scale existing automated design debugging methods, as well as new methodologies and approaches for debugging specific aspects of the verification environment. These advances help address the challenges of increased size and scope of debugging in a modern verification environment. However, there are still many more steps required in order to make debug automation widespread in an industrial flow. The next two subsections outline extensions to the contributions presented in this thesis as well as
future research directions in debug automation.

### 8.2.1 Extensions of Contributions

Abstraction and refinement methods have been one of the major contributions to both model checking and automated design debugging techniques. With respect to debugging, the two abstraction and refinement techniques presented in Chapters 3 and 4 address two critical components of debugging complexity: design size and error trace length. However, these two techniques were studied in isolation. Although the two techniques are orthogonal, future research should study how these two types of abstraction can work synergistically together. As industrial designs grow larger and their error traces longer, abstraction and refinement techniques will play a critical role in managing these aspects of debugging complexity.

The UNSAT core-based technique for managing multiple design errors in Chapter 5 is effective in utilizing UNSAT cores to reduce the debugging solution space. It is clear that UNSAT cores are directly related to correction sets [89, 110], where correction sets form the basis of solutions in a SAT-based debugging formulation. This relationship can potentially be exploited further by finding novel ways to efficiently compute UNSAT cores. As described in Chapter 5, multiple overlapping UNSAT cores are in general difficult to compute. A possible future direction for this area would be to use a heuristic to find several UNSAT cores efficiently, rather than attempt to find all of them as in [110]. Recent work have attempted to accomplish this task [108], although for the debugging use-case computational efficiency is the most important criteria since it would be used as a preprocessing step. A possible area to investigate for this debugging use-case might involve efficiently simulating several subsets of the input vector to determine if that subset is sufficient for generating the erroneous output. If it is sufficient, this means that that subset, combined with the design, can define an UNSAT core. By carefully selecting these subsets, it may be possible to efficiently find multiple overlapping UNSAT cores.

The assertion and missing assumption debugging methodologies presented in Chapters 6 and 7 both rely on models to determine the form of the correct assertion or assumption. A possible direction in this area is to study the types of errors that can occur in assertions, as well as the common types of missing assumptions. If a more accurate model of the structure of
these properties is generated, the effectiveness of these methodologies can be greatly improved.

8.2.2 Future Directions

As stated throughout this thesis, much of the work in debug automation has focused primarily on the design. However, there are many different types of issues that require debugging that do not fall narrowly within design errors. In this subsection, we outline several important problems that could benefit from debug automation, as well as other general issues related to debug automation.

Testbench Debugging

This thesis introduced two novel methodologies to debug aspects of the verification environment, namely errors within assertions and missing assumptions. However, a key aspect of the verification environment, the testbench, was not addressed. In general, testbench code can be highly complex with multiple languages, frameworks, and technologies. Although testbench code can span a huge gamut of differing technologies, solutions can be created to target individual components or technologies. For example, UVM [8] defines an object oriented framework for developing reusable testbench structures. A solution in this area could be as simple as developing a robust logging system for tracing the execution of objects. A further extension on this may even provide a tool to automatically analyze the results to aid debugging.

Another example may be to target a specific type of testbench component. Conceptual components that have loose coupling with the rest of the testbench environment such as a monitors or checkers could be prime targets for analysis. By using techniques from software model checking [75], analysis of these non-synthesizable components may be possible.

Debugging High Level Languages

The recent popularity in high-level synthesis (HLS) [25, 56] has introduced new challenges compared to traditional RTL design debugging. Industrial SoC design flows have begun to use HLS tools, where a high-level language such as C is translated into RTL. This new technology has the potential to greatly increase efficiency but, at the same time, may present many debugging challenges, some of which we list here:
• The RTL generated from the HLS is typically non-comprehensible by humans, resulting in difficulty mapping between the high-level language and the RTL.

• Bugs in the HLS tool could cause significant debugging (and verification) pain, especially considering the lack of correspondence between the two representations.

• Simulation of the high-level language and the synthesized RTL may not match. This causes a large debugging pain similar to when RTL simulation does not match gate-level simulation.

• RTL optimizations (e.g. clock gating) on high-level synthesized RTL may be impractical and extremely error prone.

These challenges open a new avenue for debugging research that will need to be addressed before HLS tools become widespread.

**Debugging Inconclusive Formal Results**

Formal verification techniques [48] are notorious for their lack of scalability to large designs. Frequently, formal tools, such as model checkers, do not return a conclusive result. Instead, they return an inconclusive result where a property cannot be fully verified for the entire state space of the design (or disproven). An example of this is bounded proof [33], where a property is proven only for a certain number of cycles. In these cases, the user of the formal tool cannot realize the full benefit of the model checking tool since he has neither fully proved or disproved his property. At this point, he may either accept the result, or try to determine why a full proof could not be found. In the latter case, an automated analysis may prove useful to aid the engineer in determining the reason why a bounded proof occurred.

**All-Solution SAT Solvers**

In the SAT-based formulation of design debugging [128], one of the critical steps is finding all satisfying solutions (with respect to some subset of important variables) to return to the user. To find all-solutions, a blocking clause is usually added to an off the shelf SAT solver. These solvers, however, are optimized for a single satisfiable solution rather than all-solutions. This
is a result of model checking being one of the most popular applications of SAT where only a SAT or UNSAT result is required. As such, there is relatively little work \cite{63} in developing all-solution SAT solvers as compared to improving off the shelf SAT solvers \cite{19}. Advances in this area could have a significant impact on SAT-based debugging techniques, as well as other niches that require all-solution SAT.

8.3 Closing Remarks

Functional debugging remains one of the predominant bottlenecks in the design flow, consuming the majority of functional verification effort. Debug automation tools have made great strides in scaling to small industrial sized designs. However, the debugging challenges have grown significantly beyond the massive size of industrial designs. Today, errors are just as likely to appear in the myriad of components that exist in the verification environment. The contributions described in this dissertation aim to expand the applicability of automation in debugging to address these two issues by introducing techniques to scale existing design debugging methods, as well as introduce two tools that address specific debugging problems in the verification environment. As outlined in this chapter, the area of functional debugging still has many challenges that need to be addressed with efficient and scalable solutions. With the continued efforts of both academia and industry, novel and innovative solutions will allow debug automation tools to increase verification efficiency and resolve this critical bottleneck in the design flow.
Bibliography


[70] Intel Corporation, “Intel Itanium Processor 9500 Series,” 


