CIRCUIT DEBUGGING WITH ERROR TRACE COMPACTION AND MAXIMUM SATISFIABILITY

by

Yibin Chen

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science Graduate Department of Electrical and Computer Engineering University of Toronto

Copyright © 2009 by Yibin Chen
Abstract

Circuit Debugging with Error Trace Compaction and Maximum Satisfiability

Yibin Chen

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2009

Improving the performance and functionality of contemporary debugging tools is essential to alleviate the debugging task. This dissertation aims at narrowing the gap between current capabilities of debugging tools and industry requirements by improving two important debugging techniques: error trace compaction and automated debugging. The first contribution, error trace compaction, leverages incremental SAT solving and heuristics to reduce the number of clock cycles required to observe a failure in an error trace. The technique presented reduces the length of the error trace to a minimum while improving performance by $8 \times$ compared to a previous technique. The second contribution uses maximum satisfiability to enhance the functionality and performance of automated debuggers. The method proposed can identify where in the design the bug is located and when in the error trace the bug is excited. Compared to a competitive SAT-based approach, our formulation produces problems that are 80% smaller and that can be solved $4.5 \times$ faster. This work also introduces two performance improvements which reduce the time to find all error sources by an additional order of magnitude.
Acknowledgements

First and foremost I wish to express my sincere gratitude to my supervisor, Professor Andreas Veneris for being a mentor and guide throughout the course of my research work. During my time at the University of Toronto, Professor Veneris has been a steady source of motivation and guidance.

Furthermore, I would like to thank my colleagues at Venssa and the University of Toronto for all their support and advice. Special thanks to Sean Safarpour, Hratch Mangassarian, Jason Luu, Brian Keng, Duncan Smith, Terry Yang, Patrick Halina, Evean Qin, and Alan Baker.

I am also indebted to my parents for being my role models and an endless source of love and encouragement throughout my academic endeavors. Many thanks to my brother and the rest of my family for providing me with a constant source of support.

I would like to thank my committee members Professors Jason Anderson, Professor Wai Tung Ng, and Professor Kostas Plataniotis for their thorough reviews of my dissertation and their insightful comments and suggestions.

Acknowledgments are due to the Natural Sciences and Engineering Research Council of Canada (NSERC) and the University of Toronto for their financial support.
# Contents

| List of Tables | v    |
| List of Figures | viii |
| List of Algorithms | ix |

## 1 Introduction

1.1 Background and Motivation ......................................................... 1

1.2 Thesis Contributions ................................................................. 6

1.2.1 Optimal Error Trace Compaction .............................................. 6

1.2.2 Automated Debugging with Maximum Satisfiability ....................... 8

1.3 Thesis Outline ................................................................. 9

## 2 Background

2.1 Introduction ................................................................. 10

2.2 Boolean Satisfiability ................................................................. 10

2.2.1 Basic Definitions and Terminology ......................................... 11

2.2.2 SAT Solver Algorithms ......................................................... 11

2.2.3 Incremental SAT ................................................................. 13

2.3 Maximum Satisfiability .......................................................... 14

2.4 Modeling Circuit Behavior in CNF .............................................. 16

2.5 Design Verification and Debug .................................................. 18
3  Optimal Error Trace Compaction

3.1 Introduction ................................................. 27
3.2 Previous Work ............................................. 28
3.3 Optimal Trace Compaction using Incremental SAT  ............................................. 29
  3.3.1 Detecting Reachability using SAT ............................................. 29
  3.3.2 Optimal Trace Compaction Algorithm ............................................. 30
  3.3.3 Reducing the Search Space for T-MUX Select Variables  ............................................. 33
3.4 Assertion based Error Trace Compaction ............................................. 35
  3.4.1 Properties with only Boolean Expressions ............................................. 36
  3.4.2 Properties with Timing Relationships ............................................. 37
  3.4.3 Properties with Repetition Specifications ............................................. 38
3.5 Summary ..................................................... 39

4  Automated Debugging with Maximum Satisfiability ............................................. 40
4.1 Introduction ................................................. 40
4.2 Previous Work ............................................. 41
4.3 Clause Level Debugging of Combinational Circuits ............................................. 43
  4.3.1 Partial MaxSat Formulation for Combinational Circuits  ............................................. 43
  4.3.2 Finding all MaxSat solutions ............................................. 46
4.4 Clause Level Debugging of Sequential Circuits ............................................. 47
  4.4.1 Partial MaxSat Formulation for Sequential Circuits ............................................. 47
  4.4.2 Extracting Temporal Information ............................................. 49
4.5 Reducing MaxSat Iterations for all Solution MaxSat ............................................. 50
  4.5.1 Finding Additional Unsat Clauses ............................................. 50
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Simple single-output gates with their respective CNF representation</td>
<td>17</td>
</tr>
<tr>
<td>5.1</td>
<td>Circuit Information for sample Designs</td>
<td>65</td>
</tr>
<tr>
<td>5.2</td>
<td>Run Time comparison of different compaction techniques</td>
<td>66</td>
</tr>
<tr>
<td>5.3</td>
<td>Compaction with and without connectors</td>
<td>68</td>
</tr>
<tr>
<td>5.4</td>
<td>Trace Compaction with SVA</td>
<td>69</td>
</tr>
<tr>
<td>5.5</td>
<td>Performance comparison between SAT and MaxSat-based debugging techniques</td>
<td>71</td>
</tr>
<tr>
<td>5.6</td>
<td>Iterative Solving Results</td>
<td>72</td>
</tr>
<tr>
<td>5.7</td>
<td>Hierarchical Debugging Results</td>
<td>76</td>
</tr>
</tbody>
</table>
## List of Figures

1.1 Simplified VLSI Design Flow ................................. 3

2.1 A Simple Example Circuit .................................... 16

2.2 Sequential Circuit and its Time Frame Expansion Model .............. 19

2.3 Typical Verification and Debug Cycle .................................. 20

2.4 Sequential circuit enhanced with correction models ...................... 22

3.1 ILA Representation of $\Phi_{\leq 4}$ .................................. 31

3.2 Example encoding for $l = 2, h = 9, and k = 10$ ............................ 34

3.3 Trace Compaction Example using only Initial and Final State ............... 35

3.4 ILA encoding with a failing property ..................................... 36

3.5 ILA with two configurations for a failing property ..................... 38

4.1 Correct and erroneous combinational circuit .............................. 44

4.2 Erroneous circuit and its ILA representation ............................. 49

4.3 Deriving solutions from substitute clauses example ....................... 54

4.4 Different implementations of module level groupings .................... 59

4.5 Multiple pass debugging example ........................................ 61

5.1 Run Time vs Original Trace Length for sample circuits .................. 67

5.2 Aggregate histograms and scatter plots for hpmc_1, mem_ctrl and mips789_1 ... 74

5.3 Group Size vs Run Time .............................................. 75
List of Algorithms

1. The DPLL algorithm ................................................................. 12
2. Unsat Cores based MaxSat algorithm ........................................ 15
3. The incremental trace compaction algorithm .............................. 32
4. The derive upper bound algorithm ........................................... 33
5. The find substitutes algorithm .................................................. 52
6. The maxsat clause debug algorithm ......................................... 55
7. The hierarchical debug algorithm ............................................. 63
Chapter 1

Introduction

1.1 Background and Motivation

Consumer electronics, such as cell phones and personal computers, have become pervasive in our modern society. At the heart of many of these systems are VLSI designs consisting of thousands or millions of logic gates. Rapid improvements in the underlying fabrication technology has driven an unprecedented increase in the performance and functionality of these circuits. However, along with these technological advancements, the complexity of digital designs in the semiconductor industry has increased dramatically. This explosion in complexity has made the use of computer tools and design methodologies essential to the VLSI development process.

In particular, CAD tools for verification and debugging have become an indispensable part of the virtually all modern design flows. Despite major advances in both industry and the research community, the time and cost invested for verification and debug remains an ongoing concern [1]. It is not uncommon that the engineering time used to ensure a design’s correctness surpasses the time spent on developing the actual design. In order to live up to the growing consumer expectation that electronic systems are free of functional errors, more powerful verification and debugging tools are necessary.

The design of digital systems today is a very complex and time consuming process that transits from the original set of specifications, through numerous phases and levels of abstraction, to the final product. Each phase broadly corresponds to an incrementally more detailed descriptions of the system. Within
this context, ensuring the correctness of each phase is a major consideration since delays due to errors can put the success and future sales forecasts of the entire product at risk. Verification must therefore occur repeatedly throughout every phase of the design process.

A simplified high-level overview of such a design flow is illustrated in Figure 1.1. Typically, the design process starts with a high-level functional description of the design specifications. This description is then translated/implemented using a Hardware Description Language (HDL) such as Verilog or VHDL. This is also known as the Register Transfer Level (RTL) description of the circuit. Depending on the results of the functional verification process, the RTL description may undergo multiple revisions before it is transitioned to the next phase. Once the RTL description passes functional verification it is synthesized into a gate level netlist. Another round of verification (usually in the form of Equivalence Checking) is then performed to ensure that the gate level netlist matches its RTL description. Once an IC layout is generated after technology mapping and place & route, the chip can be fabricated.

For the most part, the industry has automated much of the design process using tools to perform a variety of tasks such as synthesis, place & route, and technology mapping [2–4]. Nevertheless, designing a bug free circuit remains the exception rather than the norm. This is in part because the growing complexity of contemporary VLSI designs has not been accompanied by commensurate advances in verification techniques. As a result, functional verification and debugging tasks still pose a major bottleneck in the design process, and a significant portion of engineering resources are devoted to it. An often cited figure in literature is that approximately 70% of the engineering effort is spent on verification [5].

The functional verification phase of the design cycle can be roughly broken down into two major components: verification and debugging. Verification involves checking whether the RTL description of a circuit behaves according to its high-level functional description. While design simulators still play a dominant role in the verification landscape, there are also a wide variety of specialized CAD tools in use today to aid in the verification effort. Most notably, property checkers and equivalence checkers formally explore the design space to detect failures that may occur during operational corner cases [6, 7]. Functional coverage analysis tools can provide statistics and build confidence about the quality of the verification process [8, 9]. The adaptation of Assertion Based Verification (ABV) has improved the
observability of errors in the design and increased functional coverage allowing for errors to be caught earlier in the design process [10].

However, in the case that the design fails verification, localizing and rectifying the erroneous behavior, i.e. debugging, remains a predominantly manual task. A task, which not only lacks sufficient automation, but that is also made more difficult by the increase in design and verification team size and the usage of third party IP. Larger verification teams, which could be dispersed over different geographical locations, create additional communication overhead. Determining who is responsible for correcting a failure can be another perplexity; and if the error is hidden within a third party IP, designers might lack
the expertise necessary to debug the error. It is therefore not surprising that approximately 60% [10] of the verification effort is spent on debugging and the time required to identify the root cause of a failure is growing rapidly. Thus, automated debugging solutions are critical to accelerate these tasks and reduce costs.

Traditionally, automated debug solutions for hardware designs are proposed based on simulation, path tracing, and Binary Decision Diagrams (BDDs) [9, 11, 12]. While these are successful at localizing post fabrication defects, their applicability to debugging RTL designs are limited. Today, design debug still comprises of examining the error trace, analyzing the design components, and back-tracing signals. These tasks may consume a significant amount of engineering time due to long simulation traces, as well as larger and more complicated designs. Error trace compaction and automated debugging techniques seek to aid the debugging effort by producing shorter error traces and by automatically localizing potential bugs in the design. In recent years, the performance of engines based on Boolean Satisfiability (SAT) has encouraged further research in these areas [13, 14]. In this thesis we examine how these two techniques can be improved.

Error trace compaction is a technique which uses an original trace showing the existence of a bug in the circuit and constructs a shorter trace exposing the same bug. Since simulation-based verification remains the industry’s work horse, many error traces are generated through random or constrained-random simulation. In practice, these traces can span over hundreds or thousands of clock cycles. The engineering time required to track down the circumstances under which the bug occurs within these traces can be significant. Due to redundancies in the test pattern a considerably shorter trace can often be found that reproduces the same error, but in fewer clock cycles. Error trace compaction techniques such as [13, 15] show that SAT solvers can be effectively used to eliminate some of these redundancies automatically. These techniques formulate the trace compaction problem as a state reachability problem and try to find a sequence of events which can transition the circuit from an initial state to an erroneous final state. Since the error trace is at the heart of virtually every debugging effort, the performance of trace compaction algorithms can have a direct impact on debugging efficiency.

Once the trace compaction algorithm returns a shorter error trace, it can then be used by the engineer
to diagnose the error. Nevertheless, finding the root cause of a failure in a large and complex design can still be very time consuming even with the aid of trace compaction techniques. Automated debugging solutions seek to further alleviate this task by identifying a set of components (gates or modules) in the design for analysis. As with trace compaction, the most efficient of such techniques are based on Boolean Satisfiability engines [14]. In SAT-based debugging, design errors are located by adding correction models to the circuit implementation. The problem is then transformed into Boolean formula in Conjunctive Normal Form (CNF) and constrained using the expected behavior of the circuit. Solving the SAT problem with a solver implicates a set of suspect error locations which can be used to diagnose the error. Based on this formulation numerous derivative works such as [16], [17], and [18] have significantly improved the performance and scalability of automated debuggers to make them more applicable to industrial designs.

However, even though both trace compaction and automated debugging techniques have made significant progress in recent years, improving the performance and capacity of these techniques remain an ongoing challenge. This can be largely attributed to the high complexity of the task at hand. The error trace compaction problem can suffer from state space explosion similar to problems in Model Checking because a large state space must be explored to find a shorter trace. The debugging problem is a NP-complete problem whose run time can increase exponentially with the number of error locations in the design [19]. Consequently, contemporary trace compaction and automated debugging techniques still have difficulty handling the largest of industrial designs.

Furthermore, both trace compaction and automated debugging techniques still leave significant room for improvement in terms of functionality. A drawback of current trace compaction techniques is that the problem has been traditionally formulated as a state reachability problem. Existing trace compaction techniques operate under the assumption that the final state is erroneous. Consequently, most of the techniques proposed attempt to reach the final state from the initial state using fewer clock cycles. However, it is often the case that the final state of the circuit in an error trace is actually a valid state that is observable during normal operations of the design. The failure may manifest itself only if a specific sequence of events occurs. Many of the trace compaction techniques proposed are not readily applicable
to traces where the final state is not the primary indicator of the failure. As a result, the shorter trace produced might not exhibit the original failure.

On the automated debugging front, most existing debuggers only identify a set of components in the design implementation which may be erroneous. Surprisingly, none provide temporal information, that is, when during the error trace the bug is actually excited. Temporal information is very important for designers as it shows the exact circumstances under which the error occurs [20]. For trace compaction and automated design debugging to become more practical in industry, both the functionality and the performance of these techniques must be improved to keep pace with advances in VLSI technology.

1.2 Thesis Contributions

This thesis seeks to ease the debugging effort by improving contemporary trace compaction and automated debugging techniques. In the area of error trace compaction techniques this work presents an efficient compaction algorithm based on incremental SAT that preserves failing circuit properties [21]. The compacted trace is guaranteed to be of minimum length and traces with significantly fewer clock cycles can be produced.

This work also presents a novel formulation for the automated debugging problem by using Partial MaxSat to improve the performance and applicability of automated design debuggers [22]. The formulation improves the granularity of the error model allowing for a more accurate description of the error location. The proposed technique not only identifies errors in the design by their location in the code but also provides the engineer with temporal information about when the bug is excited in the error trace. We also present a set of heuristics to improve the performance of our debugger. Empirical results demonstrate the effectiveness of the techniques presented. The following subsections discuss the contributions in more detail.

1.2.1 Optimal Error Trace Compaction

The simulation trace is at the heart of almost every debugging effort as it conveys useful information about the nature of a failure. Given an error trace demonstrating a failure, simulators and waveform
viewers can provide the designer with the observability needed to identify possible discrepancies in the operations of the design. In order to debug a design, engineers typically perform a trace back from known observation points. A long trace containing many redundancies can significantly prolong this time consuming process. However, since the error traces used may be supplied by random simulation or some other verification tools, compacting the trace is generally not a trivial task. Automated tools are required to reduce the trace length.

Trace compaction techniques take as their input an error trace produced by a previously failed verification step and attempt to expose the bug within fewer clock cycles. The trace compaction method presented in this thesis guarantees that the compacted trace is of minimum length. The technique first builds a SAT instance from the Iterative Logic Array representation of the circuit. It then performs a binary search on the number of time frames in the trace such that the failure is preserved. The formulation allows for learned clauses to be kept across iterations thus reducing iterative solve time.

This thesis introduces two heuristics to reduce the time required to derive the shortened error trace. The first heuristic takes advantage of the fact that long traces can often be compacted to a fraction of the original trace length. Thus, a preprocess can bias the binary search algorithm by quickly approximating an upper bound for the minimum number of time frames in the compacted trace. This allows for a significant reduction in problem size. The second heuristic further enhances performance by adding constraints to the CNF problem such that the state space explored by solver is reduced.

In addition to the basic formulation, we also present an adaptation of trace compaction for testcases using properties specified using SystemVerilog Assertions (SVA). Since the failing properties in the original trace must be maintained in the compacted trace, we enrich our formulation with constraints to ensure property preservation. We present a CNF encoding for a subset of the SVA language and demonstrate how SVA properties can be preserved in our trace compaction formulation.

Experimental results show an average of $8 \times$ speed up in run time over a previous optimal trace compaction algorithm. Our approach also allows for considerably faster run times and scales better for longer traces while allowing for the preservation of failing properties. Simple properties can be encoded using very little overhead, increasing the size of the CNF problem by less than 1.5%.
1.2.2 Automated Debugging with Maximum Satisfiability

The second part of this thesis aims at improving the performance and functionality of automated debuggers. Given an erroneous circuit, a sequence of input values and expected output values, automated debuggers seek to find a set of error sources in the design, that if corrected can rectify the problem. For complex designs, reducing the diagnosis problem to a limited set of suspected error locations can greatly improve the productivity of engineers.

Traditionally, error sources and their corresponding correction models for automated debuggers are represented at either the gate or module level [14, 20]. In this thesis we present a method to debug circuits at the CNF clause level using a Partial MaxSat solver. Our Partial MaxSat formulation identifies clauses in the CNF representation of the circuit whose removal can rectify the error. For sequential circuits, clause-level debugging not only finds error sources according to their location in the design’s netlist (and therefore lines in the RTL code), but also identifies when the error could have been excited during the error trace. This temporal information allows the engineer to quickly identify the specific circumstances under which the bug occurred.

To ensure our method is complete, our formulation must be able to find all error locations in the design. However, a MaxSat solver returns only a single solution per invocation and therefore only finds a single error location per iteration. These iterations can consume a significant amount of CPU time. In order to reduce the time required to find all solutions this paper presents a heuristic to obtain multiple MaxSat solutions per solver invocation. This is achieved through an efficient search algorithm which derives additional MaxSat solutions based on the variable assignment returned by the solver. Since solutions can be blocked between iterations to prevent duplicates, many expensive solver invocations can be avoided.

Lastly, this work describes a method to increase the granularity of the error model using module level groupings. Groupings allow for greater flexibility in formulating the debugging problem and can help find more relevant solutions at lower error cardinalities. Module level groupings can be used to implement hierarchical debugging techniques with our formulation to decrease the search space examined by the Partial MaxSat solver.
Compared to a competitive SAT-based debugging formulation [14] our MaxSat formulation produces CNF problems that are 80% smaller and can find solutions 4.5× faster on average. The performance enhancing techniques presented further decrease the time required to find all the error locations by one order of magnitude.

1.3 Thesis Outline

The remainder of this thesis is structured as follows. In Chapter 2 some background information on Boolean Satisfiability, Maximum Satisfiability, and verification and debugging methodologies is provided. The chapter also presents a brief introduction to SystemVerilog Assertions. Chapter 3 develops an optimal error trace compaction technique using Incremental SAT. The formulation presented ensures that circuit properties are preserved when deriving a compacted trace. In Chapter 4 this work describes an automated debugging formulation using Maximum Satisfiability and provides multiple heuristics to improve the performance of our debugger. The experiments showing the effectiveness of our techniques is given in Chapter 5. Finally, Chapter 6 discusses future research directions and concludes this thesis.
Chapter 2

Background

2.1 Introduction

This chapter provides some of the relevant background material necessary for understanding the contributions of this thesis. Section 2.2 and Section 2.3 review some basic concepts relating Boolean Satisfiability and Maximum Satisfiability. Section 2.4 demonstrates how circuit behavior can be modeled using Boolean formulas in Conjunctive Normal Form (CNF). Section 2.5 contains some background information on design verification and debugging. Finally, Section 2.6 provides a brief introduction to SystemVerilog Assertions.

2.2 Boolean Satisfiability

The Boolean Satisfiability (SAT) problem has been extensively studied in literature. Historically, SAT was the first problem proven to be NP-Complete [23]. In the EDA community, SAT solvers have established themselves as all-purpose tools capable of solving a wide variety of problems. Applications include Automatic Test Pattern Generation (ATPG) [24], sequential equivalence checking [25, 26], property checking [27], timing analysis [28], routing [29], and microprocessor verification [30], just to name a few. SAT instances generated for such applications often contain millions of variables. This section briefly reviews some of the background and terminology on Boolean Satisfiability solvers and algo-
2.2.1 Basic Definitions and Terminology

The SAT problem is a decision problem, which given a propositional Boolean formula $\Phi(x_1, x_2, \ldots, x_n)$ written using only $\cdot$ (conjunction operators), $+$ (disjunction operators), $\neg$ (inverse operators), variables, and parentheses, seeks to find a variable assignment to $x_1, x_2, \ldots, x_n$ such that $\Phi$ evaluates to true. If such an assignment exists, we say that $\Phi$ is satisfiable. Otherwise a proof that no such assignment exists can be returned and $\Phi$ is said to be unsatisfiable (or UNSAT). In the context of a Boolean formula, a variable can either be assigned a value of true (1) or false (0). A literal is either a variable or the negation of a variable. A clause is a disjunction of literals.

If a propositional Boolean formula is expressed using a disjunction of clauses of one or more literals, it is said to be in Conjunctive Normal Form (CNF). An example of a propositional Boolean in CNF is listed below:

$$(\overline{x_1} + x_2 + x_3)(x_3 + \overline{x_2})(x_3)$$

In this case, $x_1, x_2,$ and $x_3$ are the variables of the formula, $(\overline{x_1} + x_2 + x_3)$ is a clause of the formula, and $\overline{x_1}, x_2,$ and $x_3$ are the literals of that clause. Clauses which only contains a single literal such as $(x_3)$ are called unit clauses.

If a propositional Boolean formula $\Phi$ in CNF is unsatisfiable, then there is a subset of the clauses $cls \subseteq \Phi$ that is also unsatisfiable. The set of clauses $cls$ is called an unsatisfiable sub-formula or unsat core of $\Phi$. A CNF formula may contain multiple unsat cores. For instance, given $\Phi = (x_1 + \overline{x_2})(\overline{x_1})(x_3 + x_4)(x_1 + x_3)$, the set of clauses $\{(x_1 + \overline{x_2}), (\overline{x_1}), (x_2)\}$ is an unsat core of $\Phi$.

2.2.2 SAT Solver Algorithms

The first SAT algorithm is often attributed to Davis and Putnam who described an algorithm to solve general SAT instances in 1960 [31]. Since then, numerous algorithms and techniques have been proposed in literature to improve the efficiency of SAT solvers. Modern SAT solvers typically require that Boolean formulas are specified in CNF. To satisfy a CNF formula, each clause needs to be be satisfied
individually. A clause is satisfied if at least one of its literals evaluates to 1. If all the literals of a clause are assigned a value of 0, the clause is said to be UNSAT. Any variable assignment which sets a clause in the CNF to UNSAT will not be able to satisfy the formula. For problem instances not presented in CNF there are algorithms such as [32] which can transform any propositional formula into a CNF formula in polynomial time. For the scope of this dissertation, we will assume that any Boolean formula not expressed in CNF is converted into CNF before given to the SAT solver.

The most prominent SAT solvers today such as Grasp [33], zChaff [34], and MiniSAT [35] are based on the algorithm proposed by Davis, Putnam, Logemann and Loveland (DPLL) [36]. While numerous advances such as efficient data structures for clause storage and Boolean Constraint Propagation (BCP), solver restarts, and conflict-driven learning have greatly improved the performance of modern solvers [34, 37, 38], the underlying framework has remained virtually unchanged. Algorithm 1 provides a high level outline of the DPLL procedure.

---

**Algorithm 1:** The DPLL algorithm

```plaintext
DPLL()
begin
    status = preprocess()
    if status ≠ UNKNOWN then return status
    while decide() do
        if deduce() = CONFLICT then
            blevel ← analyze_conflict()
            if blevel = 0 then return UNSAT
            backtrack(blevel)
        end
    end
    return SATISFIABLE
end
```

Initially, all the variables in the Boolean formula to be solved are unassigned. The solver generally performs some preprocessing using the function `preprocess()` on the CNF. If the status of preprocessing step returns UNKNOWN, i.e. it can not ascertain if the propositional formula is satisfiable or...
unsatisfiable, the main loop begins. During every iteration of the loop the function `decide()` determines if there are any unassigned variables left in the CNF problem. If there are, it picks one of those variables and assigns it either a value of 0 or 1; otherwise all the variables have been assigned a value without creating an UNSAT clause and the problem is satisfiable. The function `deduce()` performs some reasoning to determine what other assignment are implied based on the previous decision and identifies whether a conflict has occurred. A conflict occurs when different CNF clauses imply contradictory value assignments for a variable.

When a conflict is encountered, the solver must backtrack to a decision level prior to the conflict and discard the decisions. Conflict analysis finds the cause of the conflict and attempts to resolve it. During the conflict analysis process, information about the current conflict may be recorded by adding clauses to the original CNF. The added clauses do not change the satisfiability of the original formula but can often be used to prune the search space in the future and improve the performance of the solver. This mechanism is called `conflict-directed learning`. We refer to these added clauses as `learned clauses` or `conflict clauses`. The solver determines that the problem is UNSAT if a conflict is reached at decision level 0.

### 2.2.3 Incremental SAT

Generally, SAT solvers are required to determine the satisfiability of a single problem instance. However, for many practical problems such as Bounded Model Checking (BMC), similar SAT instances need to be solved repeatedly [39]. In such cases, the same conflict clauses may be relearned repeatedly over various iterations. The conflict analysis carried out to find these clauses for every iteration can be very expensive. Incremental SAT solvers maximally utilize information gathered in solving a SAT instance to solve the next structurally similar one.

For the purpose of this dissertation we are mainly interested in incremental solvers which allow us to add and remove unit clauses [40]. These solvers can retain all learned clauses across iterations. A simple explanation as to why none of the clauses are invalidated is that unit clauses effectively force a variable in the CNF problem to a specific value. The validity of learned clauses is inherently independent of
assignment decisions made by the solver. Consequently all learned clauses can be retained between two problem instances if they only differ in unit clauses. This allows for significant speed up in incremental solve time.

2.3 Maximum Satisfiability

This section reviews MaxSat and its extensions, and briefly overviews recent algorithms for MaxSat, capable of handling large complex problem instances. Given an unsatisfiable CNF formula $\Phi$, the MaxSat problems consists of identifying an assignment to the problem variables such that the number of satisfied clauses from $\Phi$ is maximized [41]. The MaxSat problem is a well-known NP-Hard optimization problem.

In the Partial MaxSat problem, the CNF formula is organized into a set of hard clauses, which must be satisfied, and a set of soft clauses, which may or may not be satisfied, i.e. $\Phi = \Phi_H \cdot \Phi_S$. For Partial MaxSat problems the objective is to find an assignment that satisfies all the hard clauses and that maximizes the number of satisfied soft clauses.

In the remainder of this dissertation, hard clauses will be represented in square brackets and soft clauses in round brackets. For example, consider the following formula:

$$\Phi = [x_1 + \overline{x_2}] [x_3] \cdot (\overline{x_1}) (x_2) (\overline{x_3} + x_1)$$

The first two clauses are hard clauses, and so must be satisfied, whereas the remaining three clauses are soft clauses and may or may not be satisfied.

In the recent past [41], the most effective MaxSat algorithms have been based on branch-and-bound (B&B), supported by effective lower bounding and dedicated inference techniques. Nevertheless, most of the experimental evaluation associated with B&B MaxSat solvers assume random and handmade problem instances, which unfortunately often bear little relationship with hard industrial instances. Recent work has addressed alternative approaches, aiming the use of MaxSat algorithms in industrial settings, and focusing on instances derived from realistic applications. The most effective algorithms are based on solving MaxSat with unsatisfiable sub-formula identification and relaxation [42-44].
These algorithms extract unsat cores from the unsatisfiable problem instance and attempt to relax (i.e. eliminate) a minimal set of clauses from those cores such that the problem becomes satisfiable. The first algorithm which relates unsat cores to the Partial MaxSat is proposed in [42] and improved in [43]. Algorithm 2 provides an outline of the technique. The algorithm first uses a SAT solver to determine if the CNF formula is UNSAT. If that is the case, it then obtains an unsat core from the solver. Unsat cores are typically a byproduct of modern SAT solvers as part of their proof of unsatisfiability. The algorithm then adds a new relaxation variable $v$ to each of the soft clauses in the unsat core. If no soft clauses are in the core (i.e. it consists of only hard clauses), the MaxSat problem is UNSAT. The function `constrain_one_hot` on line 10 constrains the number of relaxation variables that can be set to 1 to one. This process is repeated until the problem becomes satisfiable. The satisfying assignment obtained is a valid Partial MaxSat assignment to the original problem.
2.4 Modeling Circuit Behavior in CNF

Due to their versatility and ability to solve complex problems, SAT solvers have been employed extensively to tackle some of the most difficult problems in EDA. Especially in the area of formal verification, many instances of problems have been reduced to SAT. In order to take advantage of these powerful engines however, the circuit problem must first be translated into a boolean problem in CNF before they can be supplied to a solver. For most SAT-based CAD algorithms this translation can be carried out in linear time from the gate level netlist of the design [24].

To convert a gate level netlist to its CNF representation, each signal in the netlist must first be uniquely labeled. Each gate is then translated into its respective CNF representation using the mapping shown in in Table 2.1. The circuit’s CNF representation consists of the conjunction of all the clauses for every gate. Depending on the application domain and problem formulation, additional constraints may be added either at the netlist or CNF level before providing the CNF to a SAT solver.

Example 1 Figure 2.1 illustrates a sample circuit consisting of three gates. Each of the signals in the circuit are uniquely labeled. Its CNF representation is given as follows:

\[(\overline{i} + a)(\overline{j} + a)(i + j + \overline{a})\]
\[(j + \overline{b})(k + \overline{a})(\overline{j} + k + b)\]
\[(\overline{a} + c)(\overline{b} + c)(a + b + \overline{c})\]

Intuitively, each clause in the CNF representation of a gate enforces a certain portion of the truth table for that gate. For instance, for gate A, the clause \((\overline{i} + a)\) ensures that if the input \(i\) is set to 1, the
<table>
<thead>
<tr>
<th>Gate</th>
<th>Function</th>
<th>CNF</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>( y = \text{AND}(x_1, x_2, \ldots, x_n) )</td>
<td>((x_1 + \overline{y}) \cdot (x_2 + \overline{y}) \cdot \ldots \cdot (x_n + \overline{y}) \cdot \overline{x_1} + \overline{x_2} + \ldots + \overline{x_n} + y)</td>
</tr>
<tr>
<td>OR</td>
<td>( y = \text{OR}(x_1, x_2, \ldots, x_n) )</td>
<td>((\overline{x_1} + y) \cdot (\overline{x_2} + y) \cdot \ldots \cdot (\overline{x_n} + y) \cdot (x_1 + x_2 + \ldots + x_n + y))</td>
</tr>
<tr>
<td>XOR</td>
<td>( y = \text{XOR}(x_1, x_2) )</td>
<td>((\overline{x_1} + x_2 + y) \cdot (x_1 + \overline{x_2} + y) \cdot (x_1 + x_2 + y) \cdot (\overline{x_1} + \overline{x_2} + y))</td>
</tr>
<tr>
<td>NAND</td>
<td>( y = \text{NAND}(x_1, x_2, \ldots, x_n) )</td>
<td>((x_1 + y) \cdot (x_2 + y) \cdot \ldots \cdot (x_n + y) \cdot \overline{x_1} + \overline{x_2} + \ldots + \overline{x_n} + y)</td>
</tr>
<tr>
<td>NOR</td>
<td>( y = \text{NOR}(x_1, x_2, \ldots, x_n) )</td>
<td>((\overline{x_1} + \overline{y}) \cdot (\overline{x_2} + \overline{y}) \cdot \ldots \cdot (\overline{x_n} + \overline{y}) \cdot (x_1 + x_2 + \ldots + x_n + y))</td>
</tr>
<tr>
<td>XNOR</td>
<td>( y = \text{XNOR}(x_1, x_2) )</td>
<td>((\overline{x_1} + x_2 + \overline{y}) \cdot (x_1 + \overline{x_2} + \overline{y}) \cdot (x_1 + x_2 + y) \cdot (\overline{x_1} + \overline{x_2} + y))</td>
</tr>
<tr>
<td>INVERTER</td>
<td>( y = \text{INV}(x) )</td>
<td>((x + y) \cdot (\overline{x} + \overline{y}))</td>
</tr>
<tr>
<td>BUFFER</td>
<td>( y = \text{BUF}(x) )</td>
<td>((x + \overline{y}) \cdot (x + \overline{y}))</td>
</tr>
<tr>
<td>MULTIPLEXER</td>
<td>( y = \text{MUX}(e, x_1, x_2) )</td>
<td>((x_1 + \overline{y} + e) \cdot (\overline{x_1} + y + e) \cdot (x_2 + \overline{y} + \overline{e}) \cdot (\overline{x_2} + y + \overline{e}))</td>
</tr>
</tbody>
</table>

Table 2.1: Simple single-output gates with their respective CNF representation

output signal \( a \) of the gate must be set to 1. Otherwise the clause is UNSAT. Similarly, the clause \((\overline{j} + a)\) ensures that if \( j = 1 \), then \( a = 1 \); and the clause \((i + j + a)\) specifies that if \( i = 0 \) and \( j = 0 \), then \( a = 0 \). Together the clauses describe the entire truth table of the two input AND gate.

Note that a CNF representation obtained in this manner can only model combinational circuits. State elements cannot be described in CNF. However, it is possible to model the sequential behavior of circuits that contain state elements for a fixed number of clock cycles \( k \). This is achieved by replicating the combinational portion of the circuit \( k \) times and connecting the next state signal of each replica with the previous state signals of the next replica. We will use the term time frame to refer to these replicas. This construction is also known as the Iterative Logic Array (ILA) representation or Time Frame Expansion of the circuit [9].

Example 2 Figure 2.2 provides an example of a sequential circuit with two state elements. The se-
sequential circuit is given in Figure 2.2(a). The time frame expansion model of the circuit modeled over three clock cycles is given in Figure 2.2(b). Note that the output of the circuit at time frame 0 is omitted in the figure and that the output at time frame 4 is the value of $b^3$ due to the flip-flop at the output. The CNF of the circuit is given as follows:

$$
(i^1 + a^1)(j^1 + a^1)(i^1 + j^1 + a^1)
$$
$$
(j^1 + b^1)(e^0 + b^1)(j^1 + e^1 + b^1)
$$
$$
(a^1 + c^1)(b^1 + c^1)(a^1 + b^1 + c^1)
$$
$$
(i^2 + a^2)(j^2 + a^2)(i^2 + j^2 + a^2)
$$
$$
(j^2 + b^2)(c^1 + b^2)(j^2 + c^1 + b^2)
$$
$$
(a^2 + c^2)(b^2 + c^2)(a^2 + b^2 + c^2)
$$
$$
(i^3 + a^3)(j^3 + a^3)(i^3 + j^3 + a^3)
$$
$$
(j^3 + b^3)(c^2 + b^3)(j^3 + c^2 + b^3)
$$
$$
(a^3 + c^3)(b^3 + c^3)(a^3 + b^3 + c^3)
$$

To demonstrate how the above formulation models the behavior of the sequential circuit consider the case where the inputs and initial state of the time frame expanded circuit are constrained using unit clauses. Suppose that the following clauses are added to the CNF:

$$
(i^1)(j^1)(i^3)(j^3)
$$

Then the only assignment to the output variables of the circuit such that the problem is satisfied are $out^2 = 0$, $out^3 = 1$, $out^4 = 0$. These are the same values that a simulator would produce when simulating the circuit using the same sequence of inputs.

2.5 Design Verification and Debug

The goal of functional verification is to determine whether a design implementation does or does not match its specification. If the existence of an error in the design is detected, verification tools return
an error trace or counter-example that allows the engineer to reproduce the failure. The term failure specifically refers to the discrepancy between the design specification and its RTL implementation that is discovered at their corresponding outputs when given identical inputs. The term error or bug commonly refers to the root cause of a failure. Given the error trace, the purpose of debugging is to identify and correct the source of the error, once a failure is observed. For the scope of this dissertation we are only concerned with the debugging of digital circuits specified using RTL.

An overview of the typical verification and debug cycle is given in Figure 2.3. First the verification tool takes as its input the Design Under Verification (DUV), typically written in an HDL such as Verilog or VHDL, and determines if an error exists in the design. At the RTL level we distinguish between two types of verification processes: simulation-based verification and formal verification. In simulation-based verification, testbenches are used simulate the behavior of the design implementation as it would operate in the real system. Simulation-based verification is the mainstream approach for functional ver-
ification due to its scalability and ease of use. The CPU time used by a simulator is proportional to the length of the testbench and the size of the design. Hand crafted test vectors can cover major design functionalities and correctness checkers or monitors can automatically detect failures during simulation. When used in conjunction with coverage analysis [8] and random or constrained-random [45] simulation, reasonable confidence about the functional correctness of the design can be gained. Nonetheless, for large complex designs, simulation-based verification tools are only able to explore a small fraction of the design space and therefore cannot determine with certainty that the design is free of functional errors. Thus, their usefulness as push-button verification tools is still limited.

Figure 2.3: Typical Verification and Debug Cycle

Formal verification tools on the other hand, prove (or disprove) using formal reasoning that the implementation of a design matches its specification. Techniques based on engines such as SAT solvers and BDDs enable these tools to exhaustively explore the design space [30, 46, 47]. Thus, they allow the
engineer to gain absolute confidence that a specific property of the design holds under all possible circumstances. However, due to the high complexity of the verification problem, these formal verification tools are limited by the size of the circuit. As a result, formal verification techniques commonly only operate on modules or sub-systems consisting of less than half a million gates.

If the verification tool does detect a failure it returns an error trace exposing the existence of a bug. Design debugging is then used to localize, diagnose, and finally correct the bug in the RTL implementation. For the most part this is done manually with the use of structural RTL editors and by examining the error trace using waveform viewers. Automated debugging solutions seek to ease the bug diagnosis and localization process by providing a set of components that could be the cause of the failure automatically. These components may be lines of RTL code, design module instantiations, or gates in the netlist. The designer can then use these suspected error locations to devise a fix for the design in the RTL. The corrected design then undergoes another round of verification. The cycle depicted in Figure 2.3 may need to be repeated multiple times until no additional failures are discovered.

2.5.1 SAT-based Automated Debugging

A category of automated debugging tools that heavily relates to the contents of this thesis are SAT-based automated debuggers. In SAT-based debugging, the circuit is first enhanced with correction models by adding a multiplexer at the output of each gate or module. These correction models allow the solver to consider certain locations in the circuit as erroneous. The select line of the multiplexer controls whether the correction model is active or inactive. If the correction model is inactive, the circuit behaves according to its implementation. If the correction model is active, the output of the gate is left unconstrained and can be replaced with a value that can correct the error.

For sequential circuits, an ILA is constructed by unrolling the circuit for the length of the error trace before the problem is converted into CNF. Additional constraints (usually implemented using an adder) limit the number of correction models that can be active simultaneously. This number is also known as the error cardinality $N_g$ of the problem.

The circuit of Figure 2.2 enhanced with correction models and constrained by a given error cardin-
nality $N_g$ is shown in Figure 2.4. Note that a correction model is added at the output of each of the three gates for every time frame. The multiplexer select lines for a single gate are tied together across all time frames allowing the solver to treat the gate as a single error location even though it is replicated multiple times. These multiplexer select signals form the input to a three input adder. The output of the adder is constrained by the error cardinality $N_g$ to limit the number of multiplexer select lines that can be set to 1 simultaneously.

In order to localize errors in the circuit, the debugger constrains the constructed ILA with the stimulus vector and the expected output values of the error trace. Finding a satisfying assignment to the resulting formula shows which correction models must be activated such that the expected output is returned by the circuit. This effectively finds a set of functionally equivalent error locations, that if corrected could fix the bug. Sets of error locations are said to be functionally equivalent if they cannot be functionally distinguished from each other under a given stimulus trace [9]. The debugger is limited to finding the set of all functionally equivalent error locations.

Based on this formulation numerous advances to enhance the performance of debuggers have been proposed. The work from [16] presents a QBF based debugging formulation using universal quantifiers which allows for sequential circuit debugging without the need for an ILA. In [18] the concept of abstraction refinement is used to reduce the size of the SAT problem and improve performance. In [17], the authors take advantage of unsatisfiable cores to speed up the debugging process for multiple fault diagnosis problems. This approach extracts a set of unsatisfiable cores from the CNF problem and
prunes potential error locations not contained in any of the cores. A SAT-based exact debugger then finds the error locations from the reduced problem.

### 2.6 SystemVerilog Assertions

Many industrial verification flows today prominently use circuit properties to check the correctness of their designs. Properties specified using a language such as the Property Specification Language (PSL) [48] or SystemVerilog Assertions (SVA) [49] specify the design’s behavior over multiple clock cycles in a concise and unambiguous manner. PSL and SVA have grown in popularity in recent years as they can be used by both simulation and formal verification tools. In simulation-based verification, properties act as correctness checkers which report an error if any of the properties of a design are violated during simulation. In the context of formal verification flows, properties specify the correct behavior of the circuit and the main purpose of formal tools is to prove or disprove that a certain property holds in all circumstances.

Within the scope of this thesis we will use SVA syntax to specify circuit properties. SVA is an assertion specification language that is specified as part of the SystemVerilog Standard [50]. The basic building blocks of SVA are sequence and property expressions.

Sequence expressions represent a combination of simple boolean expressions that get evaluated over multiple clock cycles. In simulation, a circuit expression evaluates to true if the sequence of events it describes occurs. A subset of the sequence expression syntax is given below:

\[
\text{SEQ_EXPR} \leftarrow \text{SEQ_EXPR RANGE SEQ_EXPR | SEQ_EXPR REPETITION | (SEQ_EXPR) | EXPR}
\]

\[
\text{RANGE} \leftarrow ##r | ##[r_0 : r_1]
\]

\[
\text{REPETITION} \leftarrow [*r] | [*r_0 : r_1] | [=r] | [=r_0 : r_1] | [- > r] | [- > r_0 : r_1]
\]

\[
\text{EXPR} \leftarrow \text{EXPR} \&\& \text{EXPR | EXPR || EXPR | !EXPR | signal}
\]

The RANGE operators allow the user to specify a range of clock cycles after which the next sequence expression is checked. For instance, the sequence expression \(a ##2 b\) evaluates to true if signal \(a = 1\)
and two clock cycles later signal $b = 1$. The **repetition** operators specify how many times the sequence expression on the left hand side is repeated. Depending on the operator, repetitions must either occur consecutively (‘*’) or can be allowed to repeat intermittently (‘−’) and ‘=’). For instance, $a[*2:3]$ matches the sequence where signal $a = 1$ for either two or three consecutive clock cycles. The sequence expression $a[=2]$ matches a sequence where $a = 1$ occurs twice over any number of clock cycles. In this case there is no requirement that these two events must occur over two consecutive clock cycles. The difference between ‘−’ and ‘=’ is only relevant when these repetition operators are followed by additional signals and operators. The ‘−’ operator requires that expressions following the repetition are evaluated right after the last match whereas the ‘=’ does not. For example, $a[−2]$ ##1 $b$ requires that $b = 1$ exactly one clock cycle after the last time $a = 1$. On the other hand $a[=2]$ ##1 $b$ requires that $b = 1$ one or more clock cycles after the last time $a = 1$. For both the **range** and **repetition** operators the variable $r_1$ can be substituted by the eventuality operator ‘$’ which means that there is no upper bound on the number of clock cycle delays or repetitions.

Multiple sequences can be combined sequentially or logically to construct more complex sequences. In SVA these complex sequences are represented by property expressions. Property expressions represent circuit properties that can be verified during simulation or by formal tools. A property expression can be built from sequence expressions as follows:

$$\text{PROP_EXPR} \leftarrow \text{PROP_EXPR}$$

$$\mid \text{SEQ_EXPR} \triangleright \text{PROP_EXPR}$$

$$\mid \text{SEQ_EXPR} \trianglerightslant \text{PROP_EXPR}$$

The “$\triangleright$” and “$\trianglerightslant$” operators are both implication operators which differ only in whether the consequent expression is evaluated in the same or the next clock cycle respectively.

**Example 3** Suppose we wish to specify that if $a$ is 1 and $b$ is 1 within two to one hundred clock cycles later, then $c$ must be 1 in the next clock cycle. This assertion can be written in SVA as follows:
property p1;
    @(posedge clk) (a ##[2:100] b |=> c);
endproperty

a1: assert property(p1);

The property expression in this case is given by (a ##[2:100] b |=> c). This property is evaluated at every positive edge of the clock signal clk. The keyword assert tells the verification tool to verify the property.
2.7 Summary

In this chapter the background information relevant to this thesis is presented. Section 2.2 reviews some of the basic terminologies and algorithms for Boolean Satisfiability solvers. It also illustrates how the use of incremental SAT solvers can speed up iterative problem solving. In Section 2.3 the concepts and algorithms relevant to solving Maximum Satisfiability problems are introduced. The procedure for modeling sequential circuit behavior is given in Section 2.4. Section 2.5 provides some background material on design verification and debugging. Finally, an introduction to SystemVerilog Assertions is given in Section 2.4.
Chapter 3

Optimal Error Trace Compaction

3.1 Introduction

While adoption of formal verification techniques is on the rise, simulation based functional verification remains the industry's work horse [1]. Verification methodologies using simulation make use of test-benches composed of stimulus generators and correctness checkers or monitors. Correctness checkers identify when functional specifications are violated by the DUV. One popular way to encode circuit properties is to use SystemVerilog Assertions (SVA) [49].

When a checker identifies a violation of the specifications, the corresponding error trace is used to debug the design. The complexity of debugging depends in part on the length of the error trace that must be analyzed. The longer the trace, the more clock cycles and events must be considered.

Trace compaction reduces the length of these error traces. Trace compaction techniques generate an alternative sequence of stimulus events to transition the DUV from an initial state $S_i$ to a final state $S_f$ with fewer events or clock cycles. For simulation traces spanning thousands of clock cycles, the trace compaction problem may be broken up into smaller sub-problems to find shortcuts between intermediate states of the original trace. Many of today's trace compaction algorithms are based on heuristics that may not necessarily reduce the trace enough to ease the manual debugging effort. Thus optimal automated trace compaction techniques, which can quickly find error traces using a minimum number of clock cycles to transition the circuit from $S_i$ to $S_f$, are desirable.
Another challenge in trace compaction is that verification engineers may not only seek a shorter trace from $S_i$ to $S_f$ but they may also require that properties observed in the original trace are preserved. In today’s verification environment correctness checkers can report failures due to circuit properties specified using SVA. These properties may report failures based on a sequence of events that occurs over the duration of the error trace. A compacted trace only constrained by the initial and final state may not exhibit the original failure. Unfortunately, existing trace compaction techniques do not guarantee that properties are preserved. Instead, they re-run the simulators and rely on the checkers to confirm the validity of the trace. As a result, trace compaction techniques that can preserve sequences of events are valuable to the industry.

In this chapter, we develop a trace compaction technique to address these concerns. The main contributions are as follows. Firstly, we focus on performance by introducing an efficient optimal trace compaction algorithm which returns a compacted trace of minimum length. Our trace compaction algorithm first obtains an upper bound for the minimum trace length using SAT and then employs a binary search using incremental SAT to find an optimal trace. Secondly, we introduce the concept of assertion based error trace compaction for the purpose of property preservation. This a departure from traditional formulations as it guarantees that properties failing in the original simulation trace are preserved in the compacted trace. This allows trace compaction to be applicable to a wider variety of problems. Failing assertions specified in SVA are directly encoded into our trace compaction formulation with minimal overhead.

The remainder of the chapter is structured as follows. In Section 3.2 some of the previous advancements in this field are reviewed. Our error trace compaction algorithm is given Section 3.3. Section 3.4 illustrates how SVA can be encoded as constraints into our trace compaction formulation to preserve properties. Finally, the Chapter summary is given in Section 3.5.

### 3.2 Previous Work

Previous work in the area of trace compaction include a wide array of algorithms based on formal engines and simulation-based approaches. In [20], a collection of simulation and formal techniques
CHAPTER 3. OPTIMAL ERROR TRACE COMPACTATION

is used to iteratively reduce the error trace size. Some of the techniques presented included randomly removing a single cycle or an input event from the trace, skipping identical states, and applying Bounded Model Checking (BMC) between intermediate states. To ensure that the original error is preserved, the trace is re-simulated after each iteration.

Other techniques such as [15] and [51] use image computation to search for shortcuts between circuit states. In [51], BDDs are used to compute all one-cycle reachable states from each unique state occurring in simulation. It then creates a connected graph amongst the unique simulation states and performs Dijkstra’s algorithm to find the shortest path in the graph from the initial to the final state. Similarly [15] performs pre-image computation using an all-solution SAT solver and don’t cares to efficiently traverse the state space backwards from the final state to the initial state.

Many trace compaction techniques proposed in recent years heavily employ SAT engines. For instance, [52] uses a sequential SAT solver combined with BMC techniques to find shortcuts between intermediate states in the original trace. An optimal SAT-based algorithm which guarantees a trace of minimum length between two states is proposed in [13]. To find the minimum length trace, [13] performs a binary search on the number of intermediate time frames \( k \) between \( S_i \) and \( S_f \). This reduces the number of solved SAT instances to \( \log_2(h) \) (where \( h \) is the original trace length) whereas BMC needs to solve a linear number of SAT instances. However, this technique requires that at least one of the initial or final states is a self-transition state and, as with previous techniques, requires re-simulation of the circuit. Both [52] and [13] formulate the state reachability problem as a SAT problem by expanding the sequential circuit across \( k \) time frames.

3.3 Optimal Trace Compaction using Incremental SAT

3.3.1 Detecting Reachability using SAT

Given a sequential circuit \( C \) with initial and final state constraints \( S_i(Q) \) and \( S_f(Q) \), we can determine whether \( S_f \) can be reached from \( S_i \) in exactly \( k \) steps by constructing the Iterative Logic Array (ILA) [9] of \( C \) for \( k \) time frames (or clock cycles). The problem can be formulated as a satisfiability problem in
C HAPTER 3. O PTIMAL E RROR T RACE C OMPACTION

CNF form [24]:

\[
\Phi_{=k} = S_i(Q^0) \land (\bigwedge_{0 \leq i < k} T_c(Q^i, I^i, Q^{i+1})) \land S_f(Q^k) \tag{3.1}
\]

where \(T_c(Q, I, D)\) is \(C\)'s transition relation. \(Q, I\) and \(D\) denote the current state, input, and next state variables respectively. Note that the input variables of the circuit are unconstrained. Thus, providing \(\Phi_{=k}\) to a SAT solver will return a sequence of input signals \(I^1, I^2, \ldots, I^k\) that transitions the circuit from \(S_i\) to \(S_f\) in exactly \(k\) clock cycles.

The above formula can be extended to express the reachability analysis problem for \(\leq k\) time frames by adding multiplexers between time frames:

\[
\Phi_{\leq k} = S_i(Q^0) \land (\bigwedge_{0 \leq i < k} T_c(Q^i, I^i, D^i) \land MUX(D^i, Q^i, e^i, Q^{i+1})) \land S_f(Q^k) \tag{3.2}
\]

In the above, \(MUX(D^i, Q^i, e^i, Q^{i+1})\) represents the constraints of time frame multiplexers (T-MUXes), where \(D^i\) and \(Q^i\) are the inputs and \(e^i\) is the select variable. If \(e^i = 1\) then \(Q^{i+1} = Q^i\); otherwise \(Q^{i+1} = D^i\). These multiplexers allow the SAT solver to choose between connecting the previous state signals of each time frame to the next state signals of either the previous time frame or the time frame before the previous time frame. Intuitively, if the latter is chosen, this means that the SAT solver is removing a clock cycle from the error trace. A satisfiable solution to \(\Phi_{\leq k}\) contains an assignment to the T-MUX select variables such that \(\Phi_{\leq k}\) is satisfied. The resulting solution trace of length \(\leq k\) thus consists of the sequence of input vectors \(I^j\) for which \(e^j = 0\), for all \(0 \leq j < k\). Note that \(\Phi_{\leq k}\) is satisfiable if \(\Phi_{=k}\) is satisfiable but the converse is not necessarily true unless the initial and/or the final state are self-transition states.

Consider the formulation for \(\Phi_{\leq 4}\) given in Figure 3.1. Suppose that the minimum trace length is three but \(S_f\) cannot be reached from \(S_i\) in four time frames. Then, by setting exactly one of \(e^j\) (\(0 \leq j \leq 3\)) to 1 and the others to 0, the problem is satisfied. On the other hand this problem would be not be satisfiable if \(\Phi_{=4}\) was used.

3.3.2 Optimal Trace Compaction Algorithm

A trace from \(S_i\) to \(S_f\) of length \(m\) is a minimum length trace iff \(\Phi_{\leq m}\) is SAT and \(\Phi_{\leq m-1}\) is UNSAT. One way to solve for \(m\), given an original trace length \(h\), is to iteratively construct ILAs of different
lengths until the minimum length is achieved \[13\]. Alternatively, using our proposed construction \(\Phi_{\leq h}\), \(m\) can be solved by finding the maximum number of T-MUX select variables that can be set to 1.

Our *incremental_trace_compaction* algorithm (Algorithm 3) finds \(m\) by incrementally solving \(\Phi_{\leq h}\) using different unit constraints while leveraging the usage of learned clauses. Algorithm 3 takes the problem formulation \(\Phi_{\leq h}\) as its single argument. The function *gather_tmux_select* on line 4 gathers and returns all the T-MUX select variables while *solver.reset*\((\Phi_{\leq h})\) initializes the problem. A binary search using incremental SAT is performed by the *while* loop on line 7. Notice that the solver state is never reset in the *while* loop thus allowing learned clauses to be retained between iterations. Unit clause constraints which may vary between iterations are passed to the solver using *unit_constraints*. The *for* loop on line 11 ensures that at least \(tm\) transition relations are skipped via the T-MUXes. If the problem is satisfiable, *extract_trace* on line 15 extracts the satisfying assignment for the relevant input variables. Lines 17-19, re-adjust the upper bound, \(high\), according to the number of T-MUX select variables set to 1 in the satisfying assignment.

Due to the reuse of learned clauses between iterations, Algorithm 3 can find the minimum trace length very efficiently. Another powerful way to improve performance for traces with a minimum length \(m\) that is much smaller than the original trace length \(h\) is to derive an initial upper bound for \(m\). The *derive_upper_bound* algorithm shown in Algorithm 4 performs this as a pre-processing step to *incremental_trace_compaction*.

Algorithm 4 takes the circuit \(C\), the original trace length \(h\), and a reduction factor \(0 < f < 1\) as its arguments. The algorithm then iteratively attempts to reduce the number of effective time frames in the trace using a SAT solver until \(\Phi_{\leq n}\) becomes UNSAT. It then returns the number of time frames for which \(\Phi_{\leq n}\) was still satisfiable. The function *construct_ila*(\(\Phi, n\)) constructs the CNF represen-
Algorithm 3: The incremental_trace_compaction algorithm

**Data:** The boolean formula \( \Phi_{\leq h} \) in CNF

**Result:** The compacted trace

```plaintext
incremental_trace_compaction(\( \Phi_{\leq h} \))
```

1. \( low \leftarrow 0, high \leftarrow h \)
2. \( \{e^0 \ldots e^h\} \leftarrow \text{gather_tmux_select}(\Phi_{\leq h}) \)
3. \( \text{compacted_trace} \leftarrow \emptyset \)
4. \( \text{solver.reset}(\Phi_{\leq h}) \)
5. \( \text{while } high - low \neq 1 \text{ do} \)
6. \( \quad tm \leftarrow \lceil (high + low)/2 \rceil \)
7. \( \quad \text{unit_constraints} \leftarrow \emptyset \)
8. \( \quad // \text{Disable the last } tm \text{ time frames} \)
9. \( \quad \text{for } i = 0 \text{ to } tm - 1 \text{ do} \)
10. \( \quad \quad \text{unit_constraints} \leftarrow \text{unit_constraints} \cup (e^{h-i}) \)
11. \( \quad \text{end} \)
12. \( \quad \text{if } \text{solver.solve(unit_constraints)} \text{ then} \)
13. \( \quad \quad \text{compacted_trace} \leftarrow \text{extract_trace(solver)} \)
14. \( \quad \quad high \leftarrow tm \)
15. \( \quad \quad \text{for } j = 0 \text{ to } tm \text{ do} \)
16. \( \quad \quad \quad \text{if } \text{solver.value}(e^j) = 1 \text{ then } high \leftarrow high - 1 \)
17. \( \quad \quad \text{end} \)
18. \( \quad \text{end} \)
19. \( \quad \text{else } low \leftarrow tm \)
20. \( \text{end} \)
21. \( \text{return } \text{compacted_trace} \)
22. \( \text{end} \)
```

tation of the ILA obtained by replicating \( C \) for \( n \) time frames with all the T-MUXes added. A SAT solver is then used to test for reachability within \( \leq n \) time frames using progressively smaller \( n \) to arrive at an upper bound for the minimum trace length. The reduction factor \( f \) determines how fast the pre-processing step reduces the trace length and how close \( upper \) is to the minimum prior to calling \( \text{incremental_trace_compaction} \).
Chapter 3. Optimal Error Trace Compaction

Algorithm 4: The derive_upper_bound algorithm

Data: The circuit $C$, original trace length $h$, and a reduction factor $f$

Result: An upper bound for the minimum trace length

1. $\text{derive_upper_bound}(C, h, f)$

2. begin

3. \hspace{0.5cm} $upper \leftarrow h$

4. \hspace{0.5cm} $n \leftarrow \lceil upper \ast f \rceil$

5. \hspace{0.5cm} $\Phi_{\leq n} \leftarrow \text{construct ila}(C, n)$

6. \hspace{0.5cm} $\text{solver.reset}(\Phi_{\leq n})$

7. \hspace{1cm} while $\text{solver.solve}()$ do

8. \hspace{1.5cm} $upper \leftarrow n$

9. \hspace{1.5cm} $n \leftarrow \lceil upper \ast f \rceil$

10. \hspace{1.5cm} $\Phi_n \leftarrow \text{construct ila}(C, n)$

11. \hspace{1.5cm} $\text{solver.reset}(\Phi_{\leq n})$

12. \hspace{0.5cm} end

13. return $upper$

14. end

3.3.3 Reducing the Search Space for T-MUX Select Variables

In the previous sections, formulations and algorithms were introduced to solve the optimal trace compaction problem. However, since T-MUXes are added after every transition relation, there are many possible assignments to the T-MUX select variables that effectively result in the same number of time frame skips. For example, consider $\Phi_{\leq 4}$ again as shown in Figure 3.1, this time with a minimum trace length of 4. Here, there are no solutions of length 1, 2, or 3. In order to determine this fact, the solver must explore a state space of size $\binom{4}{1} = 4$, $\binom{4}{2}/2 = 6$, and $\binom{4}{3} = 4$, respectively.

To illustrate why these redundancies are undesirable, consider the case where the solver attempts to set $e^0 = 0$ and $e^1 = 1$, $e^2 = 1$, $e^3 = 1$. The solver will detect a conflict because a trace of length 1 does not exist. It may then attempt to set $e^1$ to be the only T-MUX select variable that is assigned a value of 0. Once again, a conflict will be detected because the new variable assignment also results in a trace length of 1.
To avoid the redundancy of the second attempt, adjacent T-MUX select variables can be tied together through connectors according to successive powers of two \[53\]. Each connector is controlled by a respective connection variables \(c^i\). If a connection variable is set to 1, adjacent T-MUX select variables must be equal; otherwise they can be set independently of each other. Using the fact that \(2^0 + 2^1 \cdots + 2^{x-1} = 2^x - 1\), the problem formulation incorporating these connectors is given as follows:

\[
\Phi_{\leq k} \land \left( \bigwedge_{0<i<k} c^i \rightarrow (e^{i-1} = e^i) \right) \land \left( \bigwedge_{0<i<k, i\neq 2^x-1} c^i \right) \land \left( \bigwedge_{0<i<k} \overline{c^i} \right) \tag{3.3}
\]

where \(x\) is any integer. The connectors group time frames together such that there is only one unique assignment to the \(e^i\)'s for a given trace length.

For incremental SAT, the implication relation in the above expression needs to be added to the \(\Phi_{\leq k}\) during construction. Adjacent T-MUXes can thus be grouped together dynamically by adding their respective connection variables as a unit clause to unit constraints in Algorithm 3. We can also take into consideration the lower bound \(low\) to further reduce the search space. An example using a lower bound length of \(low = 1\), an upper bound length of \(high = 9\), and an original length of \(h = 10\) is given in Figure 3.2. The connectors which are depicted in grey are set to 0 indicating that the adjacent T-MUX select variables are not connected. An \(x\) on a T-MUX select variable indicates that the variable is unconstrained in the SAT problem. Since we are testing whether \(S_f\) can be reached from \(S_i\) in \(\leq 9\) clock cycles the last transition relation is skipped \((e^{10} = 1)\). Similarly \(e^0 = 0\) and \(e^1 = 0\) since \(low = 1\) and traces must be of length \(\geq 2\). The remaining T-MUX select variables can be grouped together as described above. In this case, the groupings are: \(\{3\}\), \(\{4,5\}\), and \(\{6,7,8,9\}\). The SAT solver is allowed to return any solution trace whose length is within the range from 2 to 9.

![Figure 3.2: Example encoding for \(l = 2\), \(h = 9\), and \(k = 10\)](image)
3.4 Assertion based Error Trace Compaction

In the previous section, a SAT formulation is presented with the objective of deriving an error trace of minimum length from an initial state to a final state. The discussion did not address how to preserve properties that occur in the original error trace in the resulting compacted trace. Since circuit properties can define sequences of events to be maintained, using only the initial and final states to constrain a trace compaction problem is often not sufficient to ensure that properties are preserved.

For example, consider a simple sequence where the initial state request must be followed by a grant and subsequently followed by acknowledge after two clock cycles. An error trace, can include the transition from request to grant without observing an acknowledge, thus violating the property as shown in Figure 3.3(a). A traditional trace compaction technique can inadvertently find a shorter trace without ever observing grant as shown in Figure 3.3(b). In this case, the resulting trace is of no value to the verification engineer since the failing property used for debugging is not violated.

Motivated by the above example, in this section, we add to the objective of trace compaction the requirement of property preservation. This requires that properties are converted directly into CNF and added to $\Phi_{\leq k}$ as shown in Figure 3.4. Note that the final state constraint can be omitted. Converting properties into CNF is commonly used in Model Checking, however for our formulation the properties must be modified to take the effects of the T-MUXes into consideration. In the following sections we show how to encode some common SVA properties to enrich the problem formulation. The encodings given here serve as a proof of concept. A full discussion on how to encode all SVA constructs is beyond the scope of this thesis.
3.4.1 Properties with only Boolean Expressions

One of the most common assertions are boolean expressions which establish relationships that must hold for every time frame. For instance, properties with only Boolean Expressions may be used to specify that a set of registers must observe one-hot encoding or that certain states should never be reachable. Consider the assertion property:

```
property p2;
  @(posedge clk) (a && b == d);
endproperty
```

This property states that signal \( d \) must always be the conjunction of signals \( a \) and \( b \). The combinational expression can be directly converted to CNF and added to each time frame in the ILA. If this property is violated during the original error trace we need to ensure that the compacted trace also violates this property at least once. \( \Phi_{\leq k} \) thus needs to be appended with the following expression:

\[
( \bigwedge_{0 \leq i < k} \overline{p_i} \leftrightarrow (a^i \land (a^i \& \& b^i == d^i))) \land ( \bigvee_{0 \leq i < k} \overline{p_i} ) \tag{3.4}
\]

where \( a^i \), \( b^i \) and \( d^i \) are the corresponding variables for \( a \) and \( d \) in time frame \( i \) of the expanded CNF representation.

In simulation, each property is verified by creating a new instance of the property at each clock cycle and evaluating the instance starting from each time frame. Similarly we specify a property identifier \( p^i \) for each time frame \( i \) in CNF indicating whether the property evaluation starting from that time frame leads to a failure. It is set to 0 if the property is violated and the T-MUX select line \( e^i = 0 \) (i.e., the
transition relation is not skipped). At least one of the $p^i$, $0 \leq i < k$ must be set to 0 for the property to fail.

### 3.4.2 Properties with Timing Relationships

One of the most common usages of properties is to specify timing relationships between signals. For properties with timing windows we need to encode all possible clock cycle delays into CNF. As in the previous section property identifiers are used to evaluate the property instance starting at each time frame.

Consider an assertion property which states that if $a = 1$ is followed within one to three cycles by $b = 1$, then $c$ must be 1 in the next clock cycle:

**property** p3;

```plaintext
@posedge clk) (a ##[1:3] b => c);
endproperty
```

The property identifier can be encoded as:

$$\overline{p^i} \iff e^i \land e^{i+1} \land e^{i+2} \land (a^i \land b^{i+1} \rightarrow c^{i+2})$$

$$\lor (e^{i+3} \land (a^i \land b^{i+2} \rightarrow c^{i+3})) \lor (e^{i+3} \land e^{i+4} \land (a^i \land b^{i+3} \rightarrow c^{i+4}))$$

(3.5)

In order to ensure that a specific clock cycle delay is met, none of the intermediate transition relations in the sequence can be skipped. Since at least three time frames are needed to determine whether the property fails, $e^i = 0$, $e^{i+1} = 0$, and $e^{i+2} = 0$. For cases where additional clock cycles are used, the T-MUX select variables are constrained appropriately.

Note that this formulation assumes that the property fails in consecutive time frames and does not allow for intermittent time frames to be skipped. For the formulation of Section 3.3.2, minimum trace length is still guaranteed. However, when the encoding of Section 3.3.3 is used, additional constructs are required to ensure minimum trace length.

**Example 4** Consider the ILA given in Figure 3.5. For clarity we simplified the representation by omitting some T-MUXes and the connectors. Suppose that a property for the circuit can be violated in exactly
4 clock cycles starting at time frame 1. We show two possible ways to lay out the property for the error trace. In Property Encoding A this property is layed out over 4 consecutive time frames. This requires that $e_0 = 1$, $e_2 = 1$, and $e_2 = 5$ and yields no reduction in trace length. Property Encoding B on the other hand requires $e_0 = 1$, $e_2 = 0$, and $e_2 = 5$. In this case the property is disabled during time frames 2-3 allowing the the number of time frames in the error trace to be minimal.

The simplest way to adapt the property for use with the technique of Section 3.3.3 is to duplicate the property multiple times taking into consideration all possible scenarios of time frame skips. Luckily, the effective number of unconstrained for T-MUX select variables increases logarithmically with the size of the trace due to connectors. Thus the number of replications is relatively small even for longer traces.

### 3.4.3 Properties with Repetition Specifications

Consider an assertion which checks that, if $a = 1$ then $b = 1$ will repeat three times (consecutively or intermittently) followed by $c = 1$ one clock cycle after:

```vhdl
property p4;
    @(posedge clk) (a => b [- > 3] ##1 c);
endproperty
```

Since repetitions can occur intermittently there is no upper bound on the length of the sequence. Given a property identifier $p^i$ which marks the beginning of the sequence we define a set of variables $q^i_1$ marking
all possible endings of the sequence. The property fails if one \( q^i_j \) is able to demonstrate the failure. Thus \( p^i \) is given as:

\[
\overline{p^i} \leftrightarrow (a^i \land \overline{e^i}) \land \left( \bigvee_{i+3 \leq j \leq k} q^i_j \right) \tag{3.6}
\]

where

\[
q^i_j \leftrightarrow \text{SORT}(\{e^i_{b}, e^i_{b+1}, \ldots, e^i_{j-2}\}, \{s_2, s_1, s_0\})
\land (s_2 \land s_1 \land s_0) \land e^i_{j-1} \land (\overline{c^i} \land \overline{e^i}) \tag{3.7}
\]

and

\[
e^i_j \leftrightarrow (b^i \land \overline{e^i}) \tag{3.8}
\]

The \( \text{SORT} \) function constructs a sorter with three outputs \( \{s_2, s_1, s_0\} \) and inputs \( \{e^i_b, e^i_{b+1}, \ldots, e^i_{j-1}\} \). The sorter propagates all 1’s at its input to the least significant bits of its output and all 0’s to the most significant bits. The output is constrained to \( \{s_2, s_1, s_0\} = 011 \) for our example since the first two repetitions of \( b = 1 \) can occur anytime before the last repetition. The property fails if the last repetition of \( b = 1 \) is followed by \( c = 0 \) within the next clock cycle. Similar to Section 3.4.2 this encoding must be adjusted when used in conjunction with the trace compaction formulation from Section 3.3.3.

### 3.5 Summary

In this chapter, we describe an efficient trace compaction technique using incremental SAT which guarantees an optimally compacted trace. The proposed technique introduces a formulation, algorithms and heuristics to improve the run time performance. This chapter also introduces the concept of assertion based error trace compaction which uses SVA to ensure that properties are preserved from the original trace. Finally, encodings for common property types are given.
Chapter 4

Automated Debugging with Maximum Satisfiability

4.1 Introduction

Due to the risk inherent in today’s highly complex design flows, engineers are increasingly adopting automated design and verification tools to ensure correctness [54, 55]. However, once verification fails, debugging remains a predominantly manual task lacking sufficient automation. In recent years, the performance of engines based on Boolean Satisfiability (SAT) [14] have encouraged research in formal debugging techniques. The purpose of these techniques is to automatically identify the source of functional errors based on constraints specifying the circuit’s expected behavior.

However, the performance (run-time) and capacity (memory) of these techniques remain an ongoing challenge. Contemporary debuggers still have difficulty handling large industrial designs. Moreover, existing techniques are limited to identifying where the most likely error locations are in the design (spatial error locations). They do not determine when in the error trace the bug occurs (temporal error locations). Temporal information is very important for diagnosing and correcting the design [20] as it allows the designer to track down more easily when in the error trace the error occurs.

In this chapter a new approach to design debugging using Partial Maximum Satisfiability (Partial MaxSat) is presented [22]. Similar to SAT problems, Partial MaxSat problems are Boolean Satisfiability
problems expressed in CNF. However, instead of finding an assignment to a satisfiable CNF formula, MaxSat solvers find the largest satisfiable subset of CNF clauses in an unsatisfiable problem. The inverse of that subset identifies a set of error locations which could be responsible for the bug. The Partial MaxSat formulation presented in this chapter allows us to identify all error locations both spatially and temporally, while significantly reducing the size of CNF problems compared to SAT-based debugging. The technique presented in this chapter can improve the performance of debugging while providing solutions at a finer granularity.

In addition, this chapter also proposes two heuristics to improve debugger performance. First, we present a method to quickly enumerate additional Partial MaxSat solutions based on results obtained from the solver. These allow us to reduce the number of solver iterations required to find all solutions.

Next, we show how tuples of related clauses can be grouped such that they are treated as a single high-level constraint. These tuples increase the granularity of the error model and reduce the search space of the problem, improving run time. Groupings based on modules and time frames allow for greater flexibility in formulating the debugging problem and can help find more relevant solutions at lower error cardinalities. We demonstrate how groupings can be used to implement hierarchical debugging [56] with Partial MaxSat to find both module-level and gate-level error locations.

The remainder of this chapter is structured as follows. First, Section 4.2 discusses some previous contributions to the field. Then, our MaxSat formulation for combinational circuits is presented in Section 4.3. The formulation is extended to finding spatial and temporal error locations in sequential circuits in Section 4.4. An algorithm to reduce the number of MaxSat iterations to find all solutions is presented in Section 4.5. Finally, Section 4.6 uses groupings to leverage the hierarchical structure of the circuit to improve performance. Section 4.7 concludes this chapter.

4.2 Previous Work

Existing SAT-based formal techniques for design debugging can be grouped into two broad categories. Approaches that are based on satisfiability (i.e., finding a satisfying assignment to the CNF problem) and those that are based on unsatisfiability (i.e., identifying which parts of the CNF problem cannot
be satisfied). In this Section we will focus our discussion on approaches based on unsatisfiability as these approaches most closely match the work presented in this chapter. For a discussion of SAT-based debugging techniques the reader is referred to Section 2.5.1.

There are two major contributions to the field of design debugging using unsatisfiability. In [57], the first MaxSat formulation for design debug is introduced. The use of clauses for identifying exact error locations in combinational circuits is presented but deemed impractical. Instead, for sequential circuits, it is shown that MaxSat can be used as a powerful tool to group clauses together for a quick over-approximation of the solutions. Error locations are modeled at the gate-level as clauses are grouped across time frames. By increasing the granularity of errors, Safarpour et al [57] combine a groupings-based MaxSat formulation with an exact SAT-based debugger to achieve performance gains.

Furthermore, the solver [58] used by [57] significantly differs from the solution technique presented here. In [58] all satisfiable subsets are enumerated, independent of their size, using disjunctions of relaxation variables. This thesis presents an algorithm which enumerates MaxSat solutions using no relaxation variables.

In [17], unsatisfiable cores are used to speed up the debugging process for multiple fault diagnosis problems. This approach extracts a set of unsatisfiable cores from the CNF problem and prunes potential error locations not contained in any of the cores. A SAT-based exact debugger is then used to find the actual error locations from the reduced problem.

Both contributions focus on using unsatisfiability during pre-processing to improve performance. Error sources are modeled as physical locations in the design (gates or modules) and the final solution is returned by a secondary SAT-based debugger. Our approach differs from previous approaches in that we do not attempt to balance the use of unsatisfiability and satisfiability for performance gains. Instead, we use a Partial MaxSat solver on sequential circuits to find the exact error location in the design without the need for an additional solver. Our formulation models errors at a finer level of granularity offering a better resolution than other approaches in addition to being the first formulation able to locate suspects in time. Even though the search space of our problem is significantly increased, our experiments show that the impact on run time is insignificant due to advances made in modern MaxSat solvers.
4.3 Clause Level Debugging of Combinational Circuits

In this section, the MaxSat debug formulation for combinational circuits is presented. In order to express the debugging problem as a MaxSat problem, the circuit must first be converted into CNF as shown in Section 2.4. For the purpose of this paper we assume that circuits consist of single-output logic gates and CNF conversion occurs on a gate by gate basis [24]. Since the behavior of each gate is specified using a set of CNF clauses, circuit conversion occurs in linear time. The goal of our debugging formulation is to identify a tuple of clauses in the CNF representation that are most likely responsible for the failure.

4.3.1 Partial MaxSat Formulation for Combinational Circuits

The Partial MaxSat formulation to debug a combinational circuit $C$ given a correct specification is as follows:

$$\Phi = [I][O] \cdot CNF(C)$$

(4.1)

where $CNF(C)$ is the CNF representation of the erroneous circuit, $I$ represents the input constraints, and $O$ are the corresponding expected output constraints. The input and expected output constraints are modeled using hard clauses (as indicated by the square brackets) as their values are assumed to be correct. Since $I$ excites the erroneous behavior of $C$ which is eventually observed at the output, the actual output of $C$ does not match the expected output $O$. The formula $\Phi$ is therefore inherently unsatisfiable.

A MaxSat solver finds an assignment to $\Phi$ such that the set of satisfied clauses is maximized. The complement of this set represents the minimum set of clauses whose removal makes $\Phi$ satisfiable. These clauses are the most likely error sources responsible for the unsatisfiability of the problem. For the remainder of this paper this complement set will be referred to as the MaxSat solution. Each of these clauses in turn can be mapped to a set of gates that could be the cause of the erroneous behavior. Note that these clauses can originate from the same or different gates.

Example 5 Consider the circuit given in Figure 4.1. Figure 4.1(a) represents the correct circuit implementation according to the specification. In the erroneous circuit of Figure 4.1(b) the NOR gate at the
output is mistakenly implemented using an OR gate. Given the input stimulus \{i = 1, j = 1, k = 0\} the actual output of the erroneous circuit is \( c = 1 \). The expected output response is \( c = 0 \). The MaxSat debug formulation of this circuit expressed in CNF is given as follows:

\[
[i][j][\overline{k}][\overline{c}]
\]

\[
A: (i + \overline{a})(j + \overline{a})(\overline{i} + \overline{j} + a)
\]

\[
B: (\overline{k} + \overline{b})(k + b)
\]

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

For clarity, we included the gates represented by each set of clauses in the above CNF.

Since the erroneous circuit cannot produce the expected response the problem is unsatisfiable. The maximum number of clauses that can be satisfied for this problem is 10 out of 12. A Partial MaxSat solution for this problem consists of two UNSAT clauses. One of the solution sets that may be returned by a Partial MaxSat solver is:

\[
S_1 = \{C : (\overline{a} + c), C : (\overline{b} + c)\}
\]

For clarity the gate to which each of the solutions map to is given before each clause. Both clauses in \( S_1 \) correctly imply that somehow correcting gate \( C \) will fix the observed failure.

We refer to this method of debugging as clause-level debugging. The number of clauses in the solution is known as the error clause cardinality of the solution. Clause-level debugging differs from SAT-based gate-level debugging techniques in that it seeks to find a set of erroneous clauses instead of a set of erroneous gates. However, the motivation behind both of these techniques is identical. Just as
gate-level debugging finds erroneous gates in order to map them to bugs in the RTL code, clause-level errors can also be mapped to gates in the netlist or code in the RTL.

Clause-level debugging solves the debugging problem at a lower level of granularity since a single gate requires multiple clauses to specify its behavior. Intuitively this means that instead of the gate, rows in the gate’s truth table are identified as erroneous. A single erroneous gate may result in multiple UNSAT clauses in the MaxSat solution. Consequently, while the cardinality of the clause-level solution is indeed minimal, the number of gates in the corresponding gate-level solution (its gate-level cardinality) might not be minimal.

For instance, in Example 5, the solver could have alternatively returned one of three other possible solutions:

\[
S_2 = \{ A : (\bar{i} + \bar{j} + a), C : (\bar{b} + c) \} \\
S_3 = \{ C : (\pi + c), B : (k + b) \} \\
S_4 = \{ A : (\bar{i} + \bar{j} + a), B : (k + b) \}
\]

All solutions are of minimum error clause cardinality 2. However, the corresponding cardinality of gate-level solutions may vary. For \( S_2 \) (gates A and C), \( S_3 \) (gates B and C) and \( S_4 \) (gates A and B) the gate-level cardinality is 2 whereas for \( S_1 \) (only gate C) the gate-level cardinality is 1. The maximum number of UNSAT clauses due to a single gate is given by the following theorem.

**Theorem 1:** Let \( G \) be a single-output logic gate that can drive both a value of 1 and 0 at its output. Then the maximum number of clauses that can be UNSAT in \( CNF(G) \) is \( m_g - 1 \), where \( m_g \) is the number of clauses in the CNF representation of the gate.

**Proof:** Let \( y \) be the output variable of gate \( G \). In order to force \( y \) to a value of 1, \( CNF(G) \) must include a clause with literal \( y \). Similarly, to force an output value of 0, one of the clauses in \( CNF(G) \) must include the literal \( \bar{y} \). Both the literals \( y \) and \( \bar{y} \) must appear in \( CNF(G) \) at least once and they cannot appear in the same clause. Therefore, assigning any value to \( y \) causes at least one clause in \( CNF(G) \) to be satisfied. Thus, MaxSat can return a maximum of \( (m_g - 1) \) clauses per gate. ■

For instance, in Example 5, \( m_g = 3 \) for gate A. The maximum number of clauses given any variable assignment to the CNF for gate A is 2.
4.3.2 Finding all MaxSat solutions

In practice, $\Phi$ has multiple solutions of minimum cardinality but the cardinality of the solution representing the actual error location may not minimal. Conventional SAT-based debuggers find all solutions of a given error cardinality $N_g$ to ensure completeness. $N_g$ is defined as the maximum gate tuples responsible for the bug. Similarly, clause-level debuggers need to find all solutions up to a maximum cardinality $N_c$. The solver can iteratively provide these solutions but a mechanism to block previous solutions is needed to avoid duplicates.

For solutions of cardinality one (i.e., the MaxSat solution only contains one clause) converting the single clause to a hard clause effectively blocks the solution. For cardinality $m > 1$ solutions however, converting each clause to a hard clause would prevent these clauses from occurring in other solutions.

To illustrate this fact consider again Example 5. Suppose that the solution $S_2$ is found in the first MaxSat iteration. Then converting the clauses $(i + \bar{j} + a)$ and $(\bar{b} + c)$ to hard clauses would not only block $S_2$ but also the solutions $S_1$ and $S_4$ since each of these solutions contains one of those clauses. Relevant solutions might be inadvertently overlooked depending on the order in which solutions are found.

Instead, the Partial MaxSat problem must be reformulated such that at least one of the solution clauses evaluates to true. To block the set of clauses $\{Cl_1, Cl_2, \ldots, Cl_m\}$ the following hard clause must be added to the CNF problem:

$$Cl_b = [Cl_i^1 + Cl_i^2 + \ldots + Cl_i^m]$$

In the above equation, we let $Cl_i^l$ denote the sets of literals in the respective $Cl_i$ clause. Since the set of literals in $Cl_b$ is the union of literals from all the solution clauses, at least one of the clauses in $\{Cl_1, Cl_2, \ldots, Cl_m\}$ must be satisfied. For instance in Example 5 the solution $S_2$ can be blocked by adding the clause $[\bar{i} + \bar{j} + a + \bar{b} + c]$ to $\Phi$.

All solutions of cardinality $\leq N_c$ can be identified by continuously blocking solutions of minimum cardinality. Using $N_c$ we can guarantee that all solutions of a certain gate-level cardinality $N_g$ are found. The relationship between the solutions obtained when using either $N_c$ or $N_g$ for combinational circuits is given by the following theorem.
Theorem 2: Let \( m_{cl} \) be the largest value of \( m_g \) for any gate in the circuit and let \( E_g \) be the set of all MaxSat solutions of gate-level cardinality \( N_g \). Let \( E_c \) be the set of all solutions of given a maximum error clause cardinality \( N_c = N_g \cdot (m_{cl} - 1) \). Then \( E_g \subset E_c \).

Proof: By contradiction. Suppose that \( S \in E_g \) is a solution such that \( S \notin E_c \). Then \(|S| > N_g \cdot (m_{cl} - 1) \) where \(|S|\) denotes the number of clauses in \( S \). However, the maximum number of clauses that can be UNSAT for a single gate is \( m_g - 1 \) and \( m_{cl} \geq m_g \). Since \( S_c \) is of gate-level cardinality \( N_g \), \(|S| \leq N_g \cdot (m_{cl} - 1) \). Therefore \( S_g \in E_c \) contradicting our initial assumption.

For Example 5, the highest value of \( m_g \) is 3 so \( m_{cl} = 3 \). To find all gate-level solutions of cardinality 1, \( N_c \) must be 2. Note that the gate-level cardinality of the solutions can vary between \( N_c \) and \( N_g \).

4.4 Clause Level Debugging of Sequential Circuits

4.4.1 Partial MaxSat Formulation for Sequential Circuits

In this section, the MaxSat formulation for combinational circuits is extended to sequential circuits. Our formulation takes as its inputs the sequential circuit \( C \), a sequence of stimulus vectors \( I^1, I^2, \ldots, I^k \) and expected output sequence \( O^1, O^2, \ldots, O^k \). The variable \( k \) is the number of clock cycles in the error trace over which the behavior of \( C \) is modeled. The problem is also constrained by an initial state vector \( IS \). The Iterative Logic Array (ILA) of the circuit, otherwise known as the time frame expansion model, is constructed by unrolling the combinational portion of the circuit \( k \) times. This effectively translates the sequential problem into a combinational one. The ILA can then be converted into a Boolean Satisfiability instance in CNF. The input to the MaxSat solver \( \Phi \) is given by:

\[
\Phi = \prod_{i=1}^{k} [I^i][O^i] \cdot [IS] \cdot CNF(ILA_k(C))
\]

where \( ILA_k(C) \) denotes the time frame expansion of \( C \) for \( k \) time frames.

Example 6 Consider the erroneous circuit in Figure 4.2(a). The correct circuit is derived by replacing gate A with an OR gate. The ILA representation of the circuit for three clock cycles is given in Figure 4.2(b). For clarity some irrelevant gates and constraints are omitted in the figure.
The initial state of the circuit is given by $a^0 = 0$ and $b^0 = 0$. The input vectors $[i^1 = 0, j^1 = 1]$ and $[i^2 = 0, j^2 = 1]$ cause the value at the fanout of gate A to differ from the correct values in time frames 1 and 2. The effects of these two error excitations are then propagated and observed in time frame 3 where the actual output ($out^3 = 1$) of the trace differs from the expected output ($out^3 = 0$). The MaxSat solutions for $N_c = 2$ are as follows:

$$S_1 = \{A^1 : (i^1 + a^1), A^2 : (i^2 + a^2)\}$$
$$S_2 = \{A^1 : (i^1 + a^1), B^3 : (a^2 + b^3)\}$$
$$S_3 = \{A^1 : (i^1 + a^1), C^3 : (b^3 + c^3)\}$$
$$S_4 = \{B^2 : (a^1 + b^2), A^2 : (i^2 + a^2)\}$$
$$S_5 = \{B^2 : (a^1 + b^2), B^3 : (a^2 + b^3)\}$$
$$S_6 = \{B^2 : (a^1 + b^2), C^3 : (b^3 + c^3)\}$$
$$S_7 = \{C^3 : (b^3 + c^3), A^2 : (i^2 + a^2)\}$$
$$S_8 = \{C^3 : (b^3 + c^3), B^3 : (a^2 + b^3)\}$$
$$S_9 = \{C^3 : (b^3 + c^3), C^3 : (b^3 + c^3)\}$$

From these solutions, $S_1$ correctly implicates gate A as the source of the bug. The remaining solutions result from the presence of two propagation paths from the error excitations to the output signal $out^3$. One propagation path passes through the gates $A^1 \rightarrow B^2 \rightarrow C^3 \rightarrow out^3$ while the other path goes through $A^2 \rightarrow B^3 \rightarrow C^3 \rightarrow out^3$.

In the worst case the error could be excited in every clock cycle. Since a gate is replicated $k$ times in $ILA_k(C)$, an error clause cardinality of $N_c = N_g : (m_{cl} - 1) : k$ would be required to find all solutions of cardinality $N_g$. Fortunately, the number of times that the gate-level error is active, i.e. the error is excited and propagated to the output, is generally small. MaxSat finds the least number of error excitations required to observe the bug.

Unlike conventional gate-level debuggers, our MaxSat formulation is not limited to finding a set of erroneous gates for a given error cardinality. Errors from the same or different gate-level sources are not distinguished at the clause level. For instance, in Example 6 solution $S_1$ implicates two excitations
of gate A as the cause of the problem. Solution $S_2$ indicates that an excitation of gate A followed by an excitation of gate B could be the bug. In both cases the number of excitations is two but no distinction is made whether these excitations come from the same or different gates. As a result, once an error clause cardinality $N_c$ is specified, all clause-level errors are found irrespective of their corresponding gates. $N_c$ can be more appropriately specified as:

$$N_c = N_{ep} \cdot (m_{cl} - 1)$$  \hspace{1cm} (4.4)$$

where $N_{ep}$ is the maximum expected number of gate-level error excitations and propagations for a given stimulus trace.

The concept behind $N_{ep}$ is different to the one for $N_g$ that denotes the estimated number of spatial error locations that exist in the design irrespective of any error traces. Similar to $N_g$, the user can provide an estimate for $N_{ep}$ based on trace length and the complexity of the problem [14].

### 4.4.2 Extracting Temporal Information

An advantage of clause-level debugging is that both gate-level error sources and temporal bug information is provided. Since the circuit is replicated once for every clock cycle, each clause in the MaxSat
solution of $\Phi$ represents an error location both spatially and temporally. Thus, each solution clause automatically indicates when during the error trace the bug is active. For instance, in Example 6, the clause $(i^1 + a^T)$ in $S_1$ indicates that gate $A$ in time frame 1 could be one of the causes of the bug. The solution $S_1$ therefore states that gate $A$ must be active in time frames 1 and 2 to cause the error.

Another advantage of the temporal information provided is that it establishes a more specific relationship between each of the gates in the solution. In gate-level debugging, when considering higher cardinality solutions, we can only determine that the locations in the solution act together to cause the error. However, the order in which each of these error locations act is unknown. Since temporal information is included in our MaxSat formulation, we can determine the sequence in which the individual gates are related. For instance, solution $S_2$ in Example 6 indicates that an excitation of gate $A$ in time frame 1 followed by an excitation of gate $B$ in time frame 3 may be the cause of the error in the design. In contrast, SAT-based debugging would only indicate that the gates $A$ and $B$ can act together to cause the bug.

Another approach is to aggregate the error information from all MaxSat solutions by creating a histogram showing the number of suspected error sources found for each time frame. The frequency of solutions per time frame can provide hints about when the errors are excited in the trace. The error is more likely to be propagated through circuit elements close to the actual error excitation so generally the time frames with the highest frequency of solutions are the best candidates. This aggregate temporal information can help the designer analyze the error trace. Example graphs for different circuits are provided in Section 5.3 for the Experiments.

### 4.5 Reducing MaxSat Iterations for all Solution MaxSat

#### 4.5.1 Finding Additional Unsat Clauses

Since our formulation improves the granularity of the error model and finds time frame information, the search space for solutions is significantly larger compared to SAT-based debugging. Every gate is replicated over all time frames and each gate requires multiple clauses to represent. The number of MaxSat iterations required to find all solutions of cardinality $\leq N_c$ can be considerable. The purpose
of this section is to improve on the technique given in Section 4.3.2. In detail, we introduce a set of heuristics to derive more solutions based on existing MaxSat solutions that originate from previous iterations of the basic algorithm.

Consider again Example 6. The solutions $S_1$ to $S_9$ are due to the two propagation paths from the erroneous gate to the output. The respective variable assignments of each of these solutions only differ by a few variables. For instance, solution $S_1$ only differs from $S_2$ by the assignment to variable $a_2$. However, in order to find all of these solutions a total of nine MaxSat iterations is required. For larger circuits, each of these iterations can be computationally intensive.

Consider an UNSAT clause $c_{lu} \in \Phi$ for a given MaxSat assignment. Since $c_{lu}$ is UNSAT each of its literals evaluate to 0. Inverting the variable assignment for one of its literals would cause $c_{lu}$ to become satisfied. However another set of clauses $c_{ls_i} \subset \Phi$ with $|c_{ls_i}| \geq 1$ would become UNSAT. If this set only contains a single clause $c_{ls_i} = \{c_{ls}\}$ then the new assignment is another MaxSat assignment for $\Phi$. The number of UNSAT clauses in the new assignment remains the same as in the original assignment and is therefore minimal. That is $c_{lu}$ can be replaced with $c_{ls}$ in the MaxSat solution since the solution cardinality remains unchanged. We say that $c_{ls}$ is a substitute solution clause for $c_{lu}$.

An informal description of the technique applied multiple times to Example 6 is given below. Suppose the first MaxSat solution obtained is $S_1$. The clauses $(i^1 + \overline{a_1})$ and $(i^2 + \overline{a_2})$ are UNSAT. The variables $a^1$ and $a^2$ are therefore both set to 1 ($i^1$ and $i^2$ are constrained by hard clauses).

The clause $(i^1 + \overline{a_1})$ can be satisfied by setting $a^1 = 0$. As a result exactly one other clause, $(a^1 + b^2)$, becomes UNSAT. The clause $(a^1 + b^2)$ is therefore a substitute solution clause for $(i^1 + \overline{a_1})$ in $S_1$. Using this reasoning, the solution $S_4$ can be directly derived from $S_1$. Continuing from this new MaxSat assignment it is then possible to find another substitute solution clause, $(b^3 + c^3)$, by setting $b^3 = 1$. Similarly the clauses $(a^2 + b^3)$ and $(\overline{b^3} + c^3)$ can be found as substitute solution clauses for $(i^2 + \overline{a_2})$. These two sets of substitute clauses effectively identify $S_2, S_3, S_4$ and $S_7$ as additional solutions.

The algorithm to perform this local search is described in Algorithm 5. It takes an unsat solution clause $c_{lu}$, the MaxSat variable assignment $va$ and the CNF problem $\Phi$ as its inputs. The output of this algorithm is a set of substitute clause and assignment modification pairs. An assignment modification
Algorithm 5: The find_substitutes algorithm

Data: The CNF problem $\Phi$, a UNSAT clause $cl_u$, and variable assignment $va$

Result: A set of substitute clauses and modifications to $va$ for each clause

1 $find_substitutes(\Phi, cl_u, va)$
2 begin
3     stack.push([cl_u, $\emptyset$])
4     subs ← [cl_u, $\emptyset$]
5     while stack $\neq$ $\emptyset$ do
6         [cl_t, mod] ← stack.pop()
7         invert_assignment(va, mod)
8         cls_c = get_connected_clauses(\Phi, cl_t)
9         foreach literal $l \in cl_t$ do
10             $n \leftarrow 0$
11             foreach clause $cl \in cls_c$ do
12                 if sat_literals(cl, va) = {$\overline{l}$} then
13                     $n \leftarrow n + 1$
14                     $cl_s \leftarrow cl$
15                 end
16             end
17             mod_t ← mod $\cup$ $l$
18             if $n = 1$ and (subs $\cap$ [cls_s, mod_t]) = $\emptyset$) then
19                 subs ← subs $\cup$ [cls_s, mod_t]
20                 stack.push([cls_s, mod_t])
21             end
22         end
23         invert_assignment(va, mod)
24     end
25     remove_hard_clauses(subs)
26     return subs
27 end
mod consists of a subset of variables from $\Phi$. Applying a modification mod to a variable assignment means that all the variables in mod are inverted to obtain a new assignment. In this case, the assignment modifications can be applied to va to obtain new MaxSat assignments.

A stack is used by the algorithm to keep track of pairs that have not yet been examined by the algorithm. Whenever a new UNSAT clause $cl_t$ is removed from the stack, a call to invert_assignments on line 7 modifies va such that $cl_t$ is UNSAT under va. These modifications are reverted on line 23.

The search for solution clauses connected to $cl_t$ is performed within the loop on line 9. The get_connected_clauses function finds all clauses in $\Phi$ which have variables in common with $cl_t$. For each literal $l \in cl_t$ and every connected clause $cl \in cls_c$, the function sat_literals finds all satisfied literals in $cl$ given va. If $\overline{l}$ is the only satisfied literal in $cl$, then inverting the variable assignment of $l$ would convert $cl$ into an UNSAT clause. If only a single clause $cl_s \in cls_c$ becomes UNSAT, then $cl_s$ is a substitute solution clause for $cl_t$. Finally, line 25 removes any hard clauses from the result.

Note that the find_substitutes function is limited to finding MaxSat solutions which only differ by a single variable assignment from each other. Since we assume that our circuit consists of only single-output gates this method effectively finds all clauses on an error propagation path consisting of functionally equivalent error locations.

### 4.5.2 Deriving More Solutions using Substitute Clause Sets

Using the function find_substitutes a new MaxSat solution can be derived for each new substitute clause found. In this section, we examine how further solutions can be derived from combinations of substitute clauses.

Consider for example a cardinality two MaxSat solution: \{cl_{sa}, cl_{sb}\} and suppose that $cl_{ta}$ and $cl_{tb}$ are substitute clauses for $cl_{sa}$ and $cl_{sb}$ respectively. Then based on the find_substitutes algorithm it is possible to obtain the solution \{$cl_{ta}$, $cl_{sb}$\} by inverting a set of variables $mod_a$ in the MaxSat assignment. Similarly, \{$cl_{sa}$, $cl_{tb}$\} can be obtained by inverting a set of variables $mod_b$. Based on these two solutions it is likely that \{$cl_{ta}$, $cl_{tb}$\} is also a MaxSat solution and can be derived by inverting the set $mod_a \cup mod_b$. For instance, the set of cardinality 2 solutions for Example 6 is essentially the cross product of the two
The MaxSat assignment for each solution is obtained by combining their respective assignment modifications. However, solutions obtained in this manner are not guaranteed to be valid MaxSat solutions. Consider the following example.

Example 7 The CNF of the circuit given in Figure 4.3 is expressed as follows:

\[
\begin{align*}
\bar{i} \lor \bar{j} \lor \bar{c} \lor \bar{d} \lor \bar{e} \\
A: \quad (\bar{i} + \bar{a})(i + a) \\
B: \quad (\bar{j} + \bar{b})(j + b) \\
C: \quad (a + \bar{c})(\bar{a} + c) \\
D: \quad (a + \bar{d})(b + \bar{d})(\bar{a} + \bar{b} + d) \\
E: \quad (b + \bar{c})(\bar{b} + e)
\end{align*}
\]

Suppose that the initial solution is \( S_1 = \{A : (i + a), B : (j + b)\} \). Then finding the substitute clause \((\bar{a} + c)\) for \((i + a)\) by setting \(a = 1\) will yield the solution \( S_2 = \{(\bar{a} + c), (j + b)\}\). Similarly, setting \(b = 1\) will yield \( S_3 = \{(i + a), (\bar{b} + e)\}\). Setting \(a = 1\) and \(b = 1\), however, increases the number of UNSAT clauses to three since \((\bar{a} + \bar{b} + d)\) is now UNSAT.

Example 7 illustrates that combinations of substitute clauses may not be valid solutions. However, in practice most combinations of clauses derived in this manner are MaxSat solutions. To take advantage of this derivation method, we must check the validity of each derived solution. Since the number of clauses affected by the assignment modifications is generally small, the runtime required should be negligible.
The optimized clause-level debugging algorithm to find all MaxSat solutions of cardinality $N_c$ is given in Algorithm 6. After each iteration of the MaxSat solver, $\text{find substitutes}$ obtains a substitute clause set for each solution clause. The cross product of all substitute clause sets represents an overapproximation of alternative MaxSat solutions. The function $\text{remove invalid solutions}$ removes derived solutions whose assignment modifications increase the number of UNSAT clauses. On line 12 the derived solutions stored in $E_t$ are then added to the solution set $E_s$. The function $\text{block solutions}$ blocks all solutions in $E_t$ as described in Section 4.3.2. The algorithm terminates once a MaxSat solution of cardinality higher than $N_c$ is found.
4.6 Debugging using Clause Groupings

Clause-level debugging can be a very powerful technique for locating temporal and spatial error locations that are excited infrequently. However, depending on the type of bug, the error cardinality required to find the relevant solution can be large. Increasing the error cardinality can reduce debugger performance since it is exponentially related to the complexity of the debugging problem [19]. If a large number of gates in the ILA representation are responsible for the error, a higher granularity error model might be required.

This section presents a technique for increasing the granularity of our error model by grouping a set of clauses such that they are treated as a single high level constraint [57]. Since the granularity of the error model is increased when grouping multiples clause together, the search space of the MaxSat solver is generally decreased. Also, the error cardinality required to find the actual error location can be lowered if the group encompasses multiple error clauses. This allows us to increase the performance of our debugging algorithm at the cost of reducing the resolution of the results.

In this dissertation we present three different heuristics to group clauses: groupings based on gates, time frames, and modules. Assigning all the clauses belonging to the same gate to the same group effectively removes the ability of the solver to identify which entries in the truth table of the gate are responsible for the bug. Instead of erroneous clauses, the group is now identified as erroneous. However, if multiple clauses belonging to the same gate are responsible for the bug, gate-level groupings enable us to find the erroneous gate at a lower error cardinality. Similarly, bugs that are excited over many time frames can be located more readily by the solver if clauses are grouped across time frames. Time frame groupings can be used in conjunction with gate-level groupings as well as module based groupings. Module based groupings allow the solver to treat all the clauses in a circuit module as a single high level constraint. Considering that a bug in the RTL code can often translate into multiple clause level errors, module level groupings generally produce more relevant results in such cases.

Finally, this section illustrates how groupings can be used to adapt hierarchical debugging [56] for our MaxSat formulation. Hierarchical debugging iteratively reduces the size of groups by traversing the hierarchical structure of the RTL code. At each iteration, the granularity of the error model is improved
while large portions of the circuit are pruned from the search space. We show how this technique can effectively act as a preprocess for clause-level debugging to improve performance.

### 4.6.1 Creating Clause Groupings

From the MaxSat solver’s point of view, the cost of setting all the clauses in the group to be UNSAT should be the same as the cost of allowing a single clause to be UNSAT. Groupings of clauses can be created adding a single clause-selector literal to each clause in the group. To ensure that the problem remains UNSAT a unit clause consisting of the inverse of that literal is then added to $\Phi$.

Consider for instance two clauses $c_{l_1} = (l_1 + l_2 + l_3)$ and $c_{l_2} = (l_4 + l_5)$ where $c_{l_1}, c_{l_2} \in \Phi$. To create a grouping consisting of $c_{l_1}$ and $c_{l_2}$ the CNF is supplemented with an additional unit clause ($\overline{y}$).

In the new CNF $\Phi'$ $c_{l_1}$ and $c_{l_2}$ are replaced with the following clauses:

$$c'_{l_1} = (l_1 + l_2 + l_3 + y)$$

$$c'_{l_2} = (l_4 + l_5 + y)$$

Suppose $\Phi$ is satisfiable if $c_{l_2}$ and $c_{l_1}$ are removed. Then $\Phi'$ is satisfiable if the clause ($\overline{y}$) is removed since setting the $y = 1$ will allow both $c'_{l_1}$ and $c'_{l_2}$ to be satisfied.

Using this method to group connected gates, we can isolate regions in the circuit which could contain the bug. Bugs involving multiple clauses can therefore be identified without increasing the maximum cardinality. This technique effectively allows us to trade-off (i.e. improve) run time at the expense of resolution. The granularity of the error model can be modified by varying the number of gates in each of these groups.

### 4.6.2 Groupings based on Gates

One application of groupings is to include clauses belonging to the same gate in a group. This effectively sets $N_c$ to be the same as $N_{ep}$. Gate-level groupings for the circuit in Example 5 can be created as follows:

$$[i][j][\overline{a}][\overline{c}]$$

$$A: (i + a + y_a)(j + a + y_a)(\overline{i} + \overline{j} + a + y_a)$$
The variables $y_a$, $y_b$ and $y_c$ are the clause-selector variables. Removing any of the unit soft clauses will satisfy all the clauses in its respective gate.

One disadvantage of using gate-level groupings is that the \texttt{find substitutes} algorithm from Section 4.5 cannot detect additional solutions if the number of clauses in the group that would otherwise be UNSAT is larger than 1. For instance, for our above example, the first solution returned by the MaxSat solver is $(y_c)$. Using the \texttt{find substitutes} algorithm with $cl_u = (y_c)$ would yield no substitute clauses since setting $y_c = 0$ would unsatisfy more than one clause (($\overline{a} + c$) and ($\overline{b} + c$)). The benefit of gate groupings in this case is that they allow the Partial MaxSat solver to find the error location at a lower cardinality.

### 4.6.3 Groupings based on Time Frame

For sequential circuits, groupings can be created not only according to circuit structure but also across time frames. Thus, instead of creating a new group for each time frame, groups can span across fixed timing windows. The size of these timing windows represents a trade off between the accuracy of the temporal bug information and the complexity of the CNF problem. Ideally the window size should reflect the expected number of consecutive bug excitations to reduce the cardinality of the solution. If no temporal bug information is desired, a single window may encompass the entire trace.

### 4.6.4 Groupings based on Modules

A more sophisticated approach to groupings can be used for designs that are synthesized from RTL code. Hardware Description Languages (HDLs) such as Verilog and VHDL naturally group related design elements into modules or functions. Instead of identifying errors at the gate-level, module level groupings can be used to identify erroneous modules. These modules do not necessarily have to be explicit user defined modules but could also arise due to a simple add or subtract statement, or an if-else
There are multiple approaches to creating a grouping for a module. The most intuitive way is to include all the gates of a module in a group as shown in Figure 4.4(a). We can derive an alternative encoding for module-level groupings if we consider that modules in an RTL design represent implications between the input and output signals of the module. Module level debugging operates under the premise that removing the input-output relationship imposed by the module would satisfy the problem. In order to remove this relationship from the circuit however, it is not necessary to remove every clause in the module. Removing only the clauses at either the inputs or the outputs of the module will achieve the same result. The remainder of the module then can be expressed as hard clauses to reduce the search space for the MaxSat solver.

Figure 4.4(b) and (c) illustrate groupings using input and output gates respectively. Empirically, we observed that grouping the gates at the module outputs yields the best performance results. Note that while Figure 4.4 demonstrates module groupings for a combinational circuit, the technique can be easily extended to the sequential case. Time frame groupings can be used in conjunction with module groupings to improve efficiency.
4.6.5 Multiple Pass Debugging

Generally modules in digital designs are structured hierarchically with sub-modules nested inside parent modules. The simplest method to debug hierarchical designs is to create a new group for each nested module. To further boost performance, module-based hierarchical debugging [56], which iteratively traverses the module hierarchy by depth, can be used. Once a module-level solution of sufficient depth has been identified, clause-level debug can further refine the solution by identifying the exact erroneous gates and time frames.

The advantage of this hierarchical approach is two-fold. Firstly, since the intermediate results of this algorithm are suspect error modules, they can be immediately used by the designer for debug. Secondly, the search space for clause-level debugging is considerably smaller if only a few modules are being considered. Effectively, module-level debugging can act as a preprocess to speed up debugging for large circuits [17, 57]. Example 8 demonstrates how hierarchical debugging works for a circuit simulated for three clock cycles.

Example 8 Consider the circuit given in Figure 4.3. For the first iteration (Figure 4.5(a)), two groups, one for each of the two top level modules, are created. The modules are grouped across all time frames, hence no temporal information is obtained from the solutions. Using Partial MaxSat, group $G_A$ is then identified as the only solution at this level. Since we only wish to find solutions in module for $A$ in the next iteration, module $B$ is specified using only hard clauses.

Suppose now that module $A$ consists of the two sub-modules $C$ and $D$. The groups $G_C$ and $G_D$ are created as shown in Figure 4.3(b). In this iteration module $D$ is identified as erroneous. Consequently, the clauses in group $G_C$ are also converted into hard clauses. Note that the search space for each of these iterations consists of only two modules. If all modules were assigned a group on the first iteration and assuming module $B$ contains no nested modules, the search space would consist of four groupings on the first iteration and three on the second.

Figure 4.3(c), shows the case where clause-level debugging is performed for the last iteration. Since only the clauses in module $D$ are soft clauses, the solver will exclusively return solutions from that module. The last iteration also provides the temporal error locations through its solutions.
The hierarchical debugging algorithm which finds all module-level and gate-level solutions to an erroneous circuit $C$ is given in Algorithm 7. The algorithm performs a traversal of erroneous modules in the design hierarchy using a queue starting with the top level module as shown on line 5. During each iteration of the loop, the algorithm finds the nested modules $p$ to a set of erroneous modules $q$. The function $\text{module\_debug}$ on line 11 then finds all module-level solutions with groupings created only for modules in $p$. Clause-level debugging (line 17) is performed instead of module-level debugging once the module depth exceeds a certain user defined value $L$ or if the module does not contain any other sub-modules. The $\text{clause\_debug}$ function is a modified version of the $\text{maxsat\_clause\_debug}$ algorithm from Section 4.5.2 which also takes the set of modules $q$ and the circuit graph $C$ as its input.
Only the gates in $q$ are specified using soft clauses while the remainder of the circuit is expressed with hard clauses. Thus, only solutions involving clauses from $q$ are returned by the solver. Since the number of gates in the modules of $q$ are a fraction of $C$’s total number of gates, the run time per iteration is reduced.

4.7 Summary

This chapter introduces techniques for debugging circuits using Partial MaxSat. We introduce an iterative method which accurately identifies all spatial and temporal error locations in a sequential design given an error trace. Compared to SAT-based debugging our formulation improves the granularity of spatial error locations by identifying the entries in the truth table of each gate as possibly erroneous. This chapter also provides additional techniques to speed up the process of finding all MaxSat solutions. We describe a fast search algorithm that quickly identifies multiple substitute clauses and alternative solutions from a given MaxSat solution. A technique to group related clauses is presented to further improve performance using hierarchical debugging. Hierarchical debugging can quickly reduce the search space and therefore the solve time for our Partial MaxSat solver.
Algorithm 7: The hierarchical debug algorithm

Data: The circuit graph $C$, Partial MaxSat problem $\Phi$, maximum clause-level cardinality $N_c$, maximum module-level cardinality $N_m$, and maximum hierarchical depth $L$

Result: A set of module-level and clause-level error locations in the circuit

1. $hierarchical\_debug(C, \Phi, N_c, N_m, L)$

2. begin

3. module\_solns $\leftarrow \emptyset$

4. clause\_solns $\leftarrow \emptyset$

5. $t \leftarrow top\_level\_module(C)$

6. queue.enqueue($t$)

7. while queue $\neq \emptyset$ do

8. $q \leftarrow queue\_dequeue()$

9. $p \leftarrow nested\_modules(C, q)$

10. if $level(q) \leq L$ and $p \neq \emptyset$ then

11. $M \leftarrow module\_debug(C, \Phi, p, N_m)$

12. module\_soln $\leftarrow$ module\_soln $\cup$ $M$

13. foreach module $m \in M$ do

14. queue.enqueue($m$)

15. end

16. else

17. $c \leftarrow clause\_debug(C, \Phi, q, N_c)$

18. clause\_soln $\leftarrow$ clause\_soln $\cup$ $c$

19. end

20. end

21. return [module\_solns, clause\_solns]

22. end
Chapter 5

Experimental Results

5.1 Introduction

This chapter presents the results of experiments conducted for the techniques described in previous sections of this thesis. Section 5.2 provides the results for our optimal trace compaction algorithms presented in Chapter 3. Section 5.3 presents the results for the automated debugging algorithm given in Chapter 4. Finally, a results summary is given in Section 5.4.

5.2 Optimal Error Trace Compaction

In this section, we demonstrate the effectiveness of our error trace compaction technique. The techniques described in Chapter 3 are implemented in C++ and use MiniSat2 as the underlying Incremental SAT solver [35]. All experiments are run on a 64-bit Quad Core Intel CPU @ 2.00GHz with 4GB of memory. Unless otherwise stated, averages are calculated by taking the geometric mean of the results. In total, three industrial circuits and four circuits obtained from OpenCores.org [59] are evaluated. Of these, mem2wire, wb.4m8s and spi are communication cores. The circuits rsdecoder, sudoku, pipeline, and ctrl are a Reed-Solomon decoder, a sudoku solver, a data pipeline, and a traffic light controller respectively. The structural details of these circuits are shown in Table 5.1.

A comparison of the run time results of the various compaction techniques presented in this chapter
Table 5.1: Circuit Information for sample Designs

<table>
<thead>
<tr>
<th>circuit</th>
<th># gates</th>
<th># inputs</th>
<th># outputs</th>
<th># state elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem2wire</td>
<td>14758</td>
<td>41</td>
<td>59</td>
<td>1338</td>
</tr>
<tr>
<td>rsdecoder</td>
<td>11353</td>
<td>9</td>
<td>10</td>
<td>521</td>
</tr>
<tr>
<td>sudoku</td>
<td>45340</td>
<td>15</td>
<td>1</td>
<td>356</td>
</tr>
<tr>
<td>wb,4m8s</td>
<td>19273</td>
<td>567</td>
<td>708</td>
<td>250</td>
</tr>
<tr>
<td>pipeline</td>
<td>4225</td>
<td>22</td>
<td>22</td>
<td>228</td>
</tr>
<tr>
<td>spi</td>
<td>2071</td>
<td>17</td>
<td>12</td>
<td>132</td>
</tr>
<tr>
<td>ctrl</td>
<td>2152</td>
<td>15</td>
<td>4</td>
<td>12</td>
</tr>
</tbody>
</table>

The speedup over the basic binary search is given in the last column.

For instance, the circuit used for the third data row is mem2wire. The problem instance is reduced from an initial trace length of 140 to a minimal trace length of 6. Using the basic Binary Search algorithm from [13] a total of 7 SAT iterations over 1935.15 seconds is required. Using our Incremental SAT approach this run time can be reduced to 1376.35 seconds. Using our Combined approach the run time can be further reduced to only 54.67 seconds using 6 SAT iterations in total, resulting in a speedup of 35.4×.

A graphical representation of the changes in run time as the initial trace length increases is given in Figure 5.1 for the circuits mem2wire, rsdecoder, wb,4m8s, and sudoku. On average the incremental SAT technique achieves a run time improvement of 1.9×. However for circuits such as
<table>
<thead>
<tr>
<th>problem</th>
<th>Binary Search [13]</th>
<th>Incremental SAT</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>circuit</td>
<td>trace len.</td>
<td>min len.</td>
<td>iterations</td>
</tr>
<tr>
<td>mem2wire</td>
<td>20</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>mem2wire</td>
<td>80</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>mem2wire</td>
<td>140</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>mem2wire</td>
<td>200</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>rsdecoder</td>
<td>20</td>
<td>14</td>
<td>5</td>
</tr>
<tr>
<td>rsdecoder</td>
<td>80</td>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>rsdecoder</td>
<td>140</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>rsdecoder</td>
<td>200</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>sudoku</td>
<td>20</td>
<td>14</td>
<td>5</td>
</tr>
<tr>
<td>sudoku</td>
<td>80</td>
<td>14</td>
<td>7</td>
</tr>
<tr>
<td>sudoku</td>
<td>140</td>
<td>14</td>
<td>7</td>
</tr>
<tr>
<td>sudoku</td>
<td>200</td>
<td>14</td>
<td>8</td>
</tr>
<tr>
<td>wb_4m8s</td>
<td>20</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>wb_4m8s</td>
<td>80</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>wb_4m8s</td>
<td>140</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>wb_4m8s</td>
<td>200</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>pipeline</td>
<td>50</td>
<td>43</td>
<td>6</td>
</tr>
<tr>
<td>pipeline</td>
<td>200</td>
<td>43</td>
<td>8</td>
</tr>
<tr>
<td>pipeline</td>
<td>350</td>
<td>43</td>
<td>9</td>
</tr>
<tr>
<td>pipeline</td>
<td>500</td>
<td>43</td>
<td>9</td>
</tr>
<tr>
<td>spi</td>
<td>50</td>
<td>47</td>
<td>6</td>
</tr>
<tr>
<td>spi</td>
<td>200</td>
<td>47</td>
<td>8</td>
</tr>
<tr>
<td>spi</td>
<td>350</td>
<td>47</td>
<td>9</td>
</tr>
<tr>
<td>spi</td>
<td>500</td>
<td>47</td>
<td>9</td>
</tr>
<tr>
<td>ctrl</td>
<td>100</td>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td>ctrl</td>
<td>400</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>ctrl</td>
<td>700</td>
<td>16</td>
<td>9</td>
</tr>
<tr>
<td>ctrl</td>
<td>1000</td>
<td>16</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 5.2: Run Time comparison of different compaction techniques

For rsdecoder, wb_4m8s, and pipeline we see a consistent drop in performance relative to [13] as the initial trace length is increased. While pipeline is able to demonstrate an impressive speedup of more than 6× when compacting from an initial trace length of \( h = 50 \) to 43, the relative performance of the incremental algorithm decreases as the \( h \) is increased. For rsdecoder and wb_4m8s run times are
actually worse than the binary search algorithm in all cases. The run time vs original trace length graphs for the circuits mem2wire, rsdecoder, wb_4m8s, and pipeline are illustrated in Figure 5.1.

The reason for this drop can be attributed to the smaller size of the SAT problems solved by the basic binary search. For instance while Incremental SAT is solving an ILA expanded over 100 time frames, the Binary Search technique is solving an ILA with only 50 time frames. The pre-processing step from Section 3.3.2 compensates for this disadvantage. On average we were able to observe a $8 \times$ speedup in run time for the circuits tested. In part this speedup is due to the reduced size of the first SAT problems solved. Especially as $h$ is increased relative to the minimum trace length, the pre-processing step helps reduce the number of SAT instances solved while also decreasing the size of the SAT problems.

Table 5.3 compares the Incremental SAT compaction technique with and without using the connec-
tors described in Section 3.3.3. In almost all cases the heuristic allowed for a significant speedup in run time with an average improvement of almost $2.5 \times$. For ctrl a more than $6 \times$ speedup is observed.

<table>
<thead>
<tr>
<th>circuit</th>
<th>trace len.</th>
<th>min len.</th>
<th>iterations</th>
<th>run time(s)</th>
<th>iterations</th>
<th>run time(s)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem2wire</td>
<td>100</td>
<td>6</td>
<td>7</td>
<td>600.75</td>
<td>7</td>
<td>115.08</td>
<td>5.22</td>
</tr>
<tr>
<td>rsdecoder</td>
<td>100</td>
<td>11</td>
<td>6</td>
<td>248.37</td>
<td>6</td>
<td>141.82</td>
<td>1.75</td>
</tr>
<tr>
<td>sudoku</td>
<td>100</td>
<td>14</td>
<td>6</td>
<td>48.57</td>
<td>6</td>
<td>30.41</td>
<td>1.6</td>
</tr>
<tr>
<td>wb_4m8s</td>
<td>100</td>
<td>4</td>
<td>7</td>
<td>1274.35</td>
<td>7</td>
<td>1044.56</td>
<td>1.22</td>
</tr>
<tr>
<td>pipeline</td>
<td>250</td>
<td>43</td>
<td>6</td>
<td>842.3</td>
<td>8</td>
<td>411.91</td>
<td>2.04</td>
</tr>
<tr>
<td>spi</td>
<td>250</td>
<td>47</td>
<td>8</td>
<td>320.15</td>
<td>8</td>
<td>142.43</td>
<td>2.25</td>
</tr>
<tr>
<td>ctrl</td>
<td>500</td>
<td>16</td>
<td>8</td>
<td>199.23</td>
<td>9</td>
<td>30.47</td>
<td>6.54</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Average: 2.45</td>
</tr>
</tbody>
</table>

Table 5.3: Compaction with and without connectors

5.2.1 Preserving Properties

We implemented our trace compaction technique using assertions instead of final state constraints. Table 5.4 provides the size of the assertions relative to some of the problems. The number of assertion clauses and literals is given in columns 4 and 5 with the percentage overhead over the total problem size given in parentheses. The run time given is for the Incremental SAT encoding from Section 3.3.2. Note that the problems in this table are different from the problems in Table 5.2 since the circuits are modified to ensure that different assertions fail. All circuits presented in Table 5.2 use an implication operator along with Boolean expressions in their assertions statements. The problems ctrl-sva1 and pipeline-sva1 use an eventuality operator ('$') and a repetition operator ('$\rightarrow$') respectively. Compared to the total size of the CNF problem, the CNF for the assertions is relatively small. Even for the smallest circuit presented (ctrl) the SVA size does not exceed 1.5% of the total problem size.

5.3 Automated Debugging with Maximum Satisfiability

In this section we experimentally demonstrate the effectiveness of our debugging techniques. The techniques described in Chapter 4 are implemented in C++ using the solver from [43] as the underlying
## Chapter 5. Experimental Results

### Table 5.4: Trace Compaction with SVA

<table>
<thead>
<tr>
<th>circuit</th>
<th>trace len.</th>
<th>min len.</th>
<th># problem clauses</th>
<th># problem literals</th>
<th># assertion clauses</th>
<th># assertion literals</th>
<th>iterations</th>
<th>run time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>spi-sva1</td>
<td>50</td>
<td>21</td>
<td>343832</td>
<td>884264</td>
<td>342 (0.10%)</td>
<td>802 (0.09%)</td>
<td>6</td>
<td>3.44</td>
</tr>
<tr>
<td>spi-sva2</td>
<td>100</td>
<td>82</td>
<td>679036</td>
<td>1752772</td>
<td>1382 (0.20%)</td>
<td>3253 (0.19%)</td>
<td>6</td>
<td>76.23</td>
</tr>
<tr>
<td>pipeline-sva1</td>
<td>50</td>
<td>30</td>
<td>664842</td>
<td>1683984</td>
<td>5664 (0.84%)</td>
<td>13240 (0.78%)</td>
<td>6</td>
<td>18.62</td>
</tr>
<tr>
<td>pipeline-sva2</td>
<td>100</td>
<td>22</td>
<td>1355844</td>
<td>3421588</td>
<td>692 (0.05%)</td>
<td>1627 (0.05%)</td>
<td>7</td>
<td>30.78</td>
</tr>
<tr>
<td>ctrl-sva1</td>
<td>50</td>
<td>9</td>
<td>129224</td>
<td>298698</td>
<td>1832 (1.40%)</td>
<td>4288 (1.42%)</td>
<td>6</td>
<td>0.38</td>
</tr>
<tr>
<td>ctrl-sva2</td>
<td>100</td>
<td>9</td>
<td>219178</td>
<td>516956</td>
<td>692 (0.31%)</td>
<td>1627 (0.31%)</td>
<td>7</td>
<td>1.37</td>
</tr>
</tbody>
</table>

MaxSat solver. All experiments are run on a 2.20GHz Intel Core2 Duo machine with 4GB of memory. As in Section 5.2, unless otherwise stated, averages are calculated by taking the geometric mean of the results.

In total, two educational circuits (pipeline, sudoku) and eight circuits obtained from OpenCores.org [59] (divider, fpu, hpdmc, mem_ctrl, mips789, mrisc, rsdecoder, spi) are presented. A single Verilog bug is inserted into each circuit at the RTL level. These may include inverting the condition in an if-statement, changing the operator in an expression or modifying a state machine to transition to an erroneous state. In rsdecoder, for instance, the increment of a counter is changed from 1 to 2.

Once a testbench detects that a bug exists, each circuit is synthesized and converted into CNF using the method in [24]. The CNF is constrained using input and expected output values from the simulation of the correct circuit model as described in Section 4.4.1.

Table 5.5 compares the effectiveness of our clause-level (Section 4.4.1) and gate-level (Section 4.6) debugging technique against SAT-based debugging [14] with MiniSat2 [35]. The formulation of [14] most closely resembles our technique since results are returned in terms of gates. To allow for a closer comparison, gates are grouped across all time frames since results obtained in this manner are identical to the error locations obtained using [14] for a given \( N_g \).

Columns 1 to 3 give the instance of the buggy circuit, the number of gates in the design, and the number of time frames in the ILA. Columns 4 to 6 then provide the number of literals, clauses, and the run time to obtain a single solution from of the debugging algorithm of [14].
The results for our MaxSat debugging algorithm is given in columns 7-12. The first two columns give the number of literals and clauses in the MaxSat formulation. The reduction in the number of clauses compared to [14] is given in the third column. The run time results for a single MaxSat iteration is given in the next column and the speed up compared to [14] is given in column 11. The error clause cardinality given in column 12. Finally, the results for our MaxSat formulation with gate-level groupings across all time frames is given in columns 13-15.

Looking at instance \(\text{fpu}_1\), the number of gates is 86020 and the number of time frames in the error trace is 40. The problem formulation for gate-level SAT-based debugging consists of 153191 thousand literals and 60804 thousand clauses. The time it takes to solve this problem with MiniSat is 2182.57 seconds. Expressing this debugging problem using our Partial MaxSat problem we see a reduction of 83.04% in the number of clauses. The time it takes to solve this problem with Partial MaxSat is 462.12 seconds, 4.72 times faster than the SAT-based solution. The error clause cardinality is 3. Using gate-level groupings, the run time is further reduced in this case to 250.78 seconds, an 8.7 times speed up. Note also that the cardinality for this solution is reduced from 3 to 1 compared to clause-level.

Recall that for SAT-based debugging, each gate is enhanced with a correction model and constraints are included in the CNF to limit the cardinality of the solution. The size of the these constraints is quadratic with the number of gates in the circuit. Thus the number of clauses in our MaxSat formulation is considerably smaller in comparison. For the instances evaluated the size reduction is 80% on average. The geometric mean of the speed up compared to SAT is \(4.49\times\) and \(4.04\times\) for clause-level and gate-level debugging respectively.

The number of clauses and literals for gate-level MaxSat is not given in the table but can be easily calculated. From Section 4.6, each clause in the group will have one additional literal. Each gate in the design will belong to its own group and one new unit clause is added for each gate since the group spans all time frames. Therefore, the number of literals is increased by the number of clauses and the number of gates in the design from clause-level formulation. The number of clauses is only increased by the number of gates in the design.
## Table 5.5: Performance comparison between SAT and MaxSat-based debugging techniques

<table>
<thead>
<tr>
<th>Error Trace</th>
<th>SAT-Based Debugging</th>
<th>Clause-Level Debugging</th>
<th>Gate-Level Groupings</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># gates</td>
<td># time</td>
<td># lits</td>
</tr>
<tr>
<td>divider_1</td>
<td>6291</td>
<td>40</td>
<td>9454k</td>
</tr>
<tr>
<td>divider_2</td>
<td>6291</td>
<td>40</td>
<td>9641k</td>
</tr>
<tr>
<td>fpu_1</td>
<td>86020</td>
<td>40</td>
<td>153191k</td>
</tr>
<tr>
<td>fpu_2</td>
<td>87144</td>
<td>19</td>
<td>6013k</td>
</tr>
<tr>
<td>hpmc_1</td>
<td>18444</td>
<td>28</td>
<td>4677k</td>
</tr>
<tr>
<td>hpmc_2</td>
<td>18444</td>
<td>58</td>
<td>7464k</td>
</tr>
<tr>
<td>mem_ctrl_1</td>
<td>55174</td>
<td>40</td>
<td>92935k</td>
</tr>
<tr>
<td>mem_ctrl_2</td>
<td>55174</td>
<td>40</td>
<td>96131k</td>
</tr>
<tr>
<td>mips789_1</td>
<td>73600</td>
<td>32</td>
<td>131773k</td>
</tr>
<tr>
<td>mips789_2</td>
<td>38524</td>
<td>158</td>
<td>8569k</td>
</tr>
<tr>
<td>masic_1</td>
<td>22452</td>
<td>42</td>
<td>40694k</td>
</tr>
<tr>
<td>pipeline_1</td>
<td>5843</td>
<td>181</td>
<td>24776k</td>
</tr>
<tr>
<td>pipeline_2</td>
<td>6318</td>
<td>69</td>
<td>14377k</td>
</tr>
<tr>
<td>rsdecoder_1</td>
<td>15738</td>
<td>50</td>
<td>31771k</td>
</tr>
<tr>
<td>rsdecoder_2</td>
<td>15732</td>
<td>100</td>
<td>49634k</td>
</tr>
<tr>
<td>spi_1</td>
<td>3427</td>
<td>14</td>
<td>3005k</td>
</tr>
<tr>
<td>spi_2</td>
<td>3357</td>
<td>143</td>
<td>13697k</td>
</tr>
<tr>
<td>sudoku_1</td>
<td>46668</td>
<td>61</td>
<td>102302k</td>
</tr>
</tbody>
</table>

Average: 79.92%  Average: 4.49  Average: 4.04
5.3.1 All Solution Partial MaxSat

Sections 4.3.2 and 4.5 discuss how all solutions for a given cardinality can be obtained. Table 5.6 summarizes the results for seven of our sample instances. Columns 2-4 give the number of MaxSat iterations, the number of solutions found and the total run time required to find all solutions. The average of the number of solutions per iteration and the run time per iteration are given in column 5 and 6 respectively.

Considering all the circuits in Table 5.6, the number of solutions found per iteration is 8. The time required to run the `find_substitutes` algorithm and verify the cross-product of substitute sets is negligible. Since each solution would have required a separate MaxSat iteration this translates into a reduction in total run time by 91%. Our method is effective since errors are often propagated to a single fanout gate. In other cases, the value of the output can be changed by manipulating a single input without affecting the remainder of the circuit. For bugs with lower error clause cardinality and where the error propagates to multiple gates in its proximity, the method can be less effective.

In our best case, all 68 solutions of `fpu_1` are found using only 3 MaxSat iterations. However, one reason why the number of solutions in this instance is large compared to other instances is related to the cardinality of the MaxSat solution. With an error clause cardinality of three, many of the solutions for `fpu_1` are different combinations of the same gates. The actual number of distinct error locations for `fpu_1` is 29.

Table 5.6: Iterative Solving Results

<table>
<thead>
<tr>
<th>design</th>
<th># iter</th>
<th># soln</th>
<th>total time</th>
<th>soln/iter</th>
<th>time/iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>divider_1</td>
<td>10</td>
<td>65</td>
<td>111.22</td>
<td>6.5</td>
<td>11.12</td>
</tr>
<tr>
<td>fpu_1</td>
<td>3</td>
<td>68</td>
<td>1088.68</td>
<td>22.67</td>
<td>362.89</td>
</tr>
<tr>
<td>hpdmc_1</td>
<td>8</td>
<td>26</td>
<td>30.82</td>
<td>3.25</td>
<td>3.85</td>
</tr>
<tr>
<td>mem_ctrl_1</td>
<td>8</td>
<td>14</td>
<td>231.7</td>
<td>1.75</td>
<td>28.96</td>
</tr>
<tr>
<td>mips789_1</td>
<td>33</td>
<td>312</td>
<td>3049.83</td>
<td>9.45</td>
<td>92.42</td>
</tr>
<tr>
<td>pipeline_1</td>
<td>21</td>
<td>195</td>
<td>825.64</td>
<td>9.29</td>
<td>39.32</td>
</tr>
<tr>
<td>rsdecoder_1</td>
<td>2</td>
<td>2</td>
<td>49.68</td>
<td>1</td>
<td>24.84</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>8.02</strong></td>
<td><strong>32.78</strong></td>
</tr>
</tbody>
</table>

Average:
5.3.2 Visualizing Temporal Information

As described in Section 4.4.2, providing temporal debug information is crucial in design debugging. The aggregate temporal information extracted by our technique for the circuits mips789, hpdmc, and mem_ctrl is illustrated in Figure 5.2. The frequency each time frame is implicated by a solution clause is shown in the histograms in Figure 5.2. The likelihood of an error being active during the time frame is indicated by the height of the bars. The scatter plots next to the histograms plot error locations vs. the time frame for which they are found. The purpose of these is to illustrate which error locations are implicated multiple times for different time frames by the debugger. The y-axis lists all unique error locations found by the algorithm and the x-axis shows the time frames during which these locations can be excited to cause the error.

For instance, the bug in hpdmc is created by removing a signal assignment in the RTL of a state machine. The actual error excitation occurs in time frame 24 since the removed signal was not assigned the proper value. The graph shows three highly likely erroneous time frames (23 to 25) and two less probable ones (20 to 21). For this case, a total of 20 distinct error locations are found. Many error locations in time frame 24 are also solutions in time frame 25.

For mem_ctrl we inverted one of the select signals of a MUX that controlled the data path. The solutions indicate that the bug could be fixed in time frame 34 when the error is excited before the data is propagated to the output. Since the data propagates through different gates in each time frame the solution gates are mostly unique as indicated in Figure 5.2(b). As the error propagates through the datapath the number of possible error sources also decreases. These graphs can allow the engineer to focus on specific regions in the design during specific time frames to correct the problem.

It is not always the case, however, that the height of the histogram provides the actual error excitation. The bug for mips789 was created by changing the default assignment of a signal to a wrong value. From the scatter plot of Figure 5.2(c) we see that some error locations are implicated multiple times by the debugger until time frame 19. In this case examining these suspected locations instead of time frame 19 would be more beneficial. Nevertheless, the shape of the graph can provide valuable information to the engineer about the nature of the problem.
Figure 5.2: Aggregate histograms and scatter plots for hpdmc, mem, and mips789.
5.3.3 Grouping Clauses

Figure 5.3 depicts the average run time given different group sizes for all the circuits in Table 5.5. Group sizes are given in terms of the number of gates in the group and groups span across all time frames. All the gates in the group are connected but groupings are created randomly. Figure 5.3 shows that the average run time generally decreases as the size of the group increases. The largest performance gain occurs when increasing the group size to 50 gates per group but the marginal gain is relatively small when increasing the size of groups further.

![Figure 5.3: Group Size vs Run Time](image)

5.3.4 Hierarchical Debugging

Table 5.7 summarizes the results of our hierarchical debugging algorithm from Section 4.6.5. The purpose of these results is to compare the performance of clause-level debug with and without a hierarchical search. We only consider the time to find the actual erroneous module and time to find all clause-level error locations within that module. The number of gates in the targeted module is given in column 2. The time required to find the module using groupings as described in Section 4.6.4 is given in column 3.

In our best performing example (fpu_1) the search space is reduced to 8320 gates (10% of the total gate count) and the search time is reduced by 5.4×. In all instances, the reduction in run time is
sufficient to compensate for the additional time required to find the module. Comparing the average time per iteration from Table 5.7 against Table 5.6, a speed up of $1.56 \times$ is observed.

Table 5.7: Hierarchical Debugging Results

<table>
<thead>
<tr>
<th>design</th>
<th>Module Level Debug</th>
<th>Clause Level Debug of Module</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># gates</td>
<td>search time</td>
</tr>
<tr>
<td>divider_1</td>
<td>5384</td>
<td>6.35</td>
</tr>
<tr>
<td>fpu_1</td>
<td>8320</td>
<td>2.89</td>
</tr>
<tr>
<td>hpdmc_1</td>
<td>6930</td>
<td>3.88</td>
</tr>
<tr>
<td>mem_ctrl_1</td>
<td>11035</td>
<td>25.24</td>
</tr>
<tr>
<td>mips789_1</td>
<td>66240</td>
<td>9.27</td>
</tr>
<tr>
<td>pipeline_1</td>
<td>1578</td>
<td>12.33</td>
</tr>
<tr>
<td>rsdecoder_1</td>
<td>8499</td>
<td>60.4</td>
</tr>
</tbody>
</table>

Average: 21.04 | 1.56

5.4 Summary

The experimental results for error trace compaction show that our trace compaction algorithm can effectively derive an error trace of minimum length while preserving failing properties. Our formulation scales considerably better compared to a previous optimal technique; and for the experiments conducted we were able to observe an $8 \times$ improvement in run time performance on average. The number of CNF clause added to encode the failing properties do not exceed $1.5\%$ of the total problem size.

For automated debugging, our results show that our approach can find a single error location $4.5 \times$ faster on average. The CNF problems constructed are $80\%$ smaller compared to a gate-level SAT-based approach. As a result our formulation requires a significantly lower memory footprint compared to the SAT-based formulation. This section also shows how time frame information can be organized to provide crucial debug information to the engineer. When finding all solutions, the optimization techniques presented allow us to identify all MaxSat solutions using $87.5\%$ fewer solver iterations while improving the run time per iteration by $1.6 \times$. 
Chapter 6

Conclusion and Future Work

6.1 Contributions

This thesis seeks to reduce the engineering effort devoted to debugging by improving contemporary error trace compaction and automated debugging techniques. On the topic of error trace compaction techniques, this work presents an efficient compaction algorithm using Incremental SAT. The length of the compacted trace is guaranteed to be minimal and traces with significantly fewer clock cycles can be produced. The proposed technique introduces a formulation, algorithms and heuristics to improve the average run time performance by $8 \times$ on average compared to a previous optimal compaction technique. This dissertation further enhances the applicability of trace compaction algorithms by adding the requirement of property preservation to compacted error traces. Property preservation ensures that failing properties in the original trace also fail in the compacted trace. This allows our technique to be extended to error traces where circuit properties are used to detect the existence of bugs in the design.

This work also presents a novel formulation of the debugging problem using Partial MaxSat. Our formulation not only provides a more accurate description of the error locations by introducing clause-level debugging, but also provides the engineer with temporal information about when the bug is excited in the error trace. The omission of a correction model in our formulation allows for a significantly smaller problem formulation and better run time performance compared to a competitive SAT-based approach. In order to further improve debugger performance, this work presents two heuristics. The
first one reduces the number of solver iterations when finding all solutions by introducing a fast search algorithm which can quickly find additional MaxSat solutions after each solver invocation. The second one uses hierarchical debugging to reduce the solver search space thus reducing solve time per iteration. For clause-level debugging we observe a $4.5 \times$ improvement in run time and a 80% reduction in problem size compared to a competitive SAT-based approach. Our search algorithm can find 8 additional MaxSat solution per iteration with little computational overhead. Hierarchical debugging further improves the performance of clause-level debug by focusing on specific modules in the design and reducing the search time per iteration by $1.56 \times$.

6.2 Future Work

Although there are many tools on the market to reduce verification costs, most of them focus on the detection of errors and few of them are concerned with reducing the debugging search space. To that end, the work presented in this thesis addresses two major areas of improvement: error trace compaction and automated debugging. In the following, we will discuss possible extensions and research directions for these topics.

The trace compaction problem has been extensively studied in literature but its scope has been limited to a form of reachability analysis. For error trace compaction, the formulation presented here is the first effort to take property preservation into consideration. However, we only cover a small portion of the SVA syntax and for our experiments we focused on encoding a single property at a time so the size of the properties remains small relative to circuit size. If additional SVA language features are supported and multiple properties are encoded for a single error trace, more efficient implementations for property preservation will be required due to memory limitations. An alternative approach that might be more effective than the technique presented here, is to convert the SVA property into an circuit with a state machine. This would allow the size of the properties to increase only linearly with trace length.

While this dissertation addressed the issue of performance, the work required to make compaction techniques more applicable to industrial applications remains extensive. For instance, traces used for testing can span over millions of clock cycles and fitting ILAs of such lengths into memory is impossible
for large designs. More effective heuristics and the usage of a divide-and-conquer approach for longer trace are required. One possibility is to split the error trace into multiple segments and compact each segment individually. The difficulty lies in preserving properties that may need to span across multiple segments.

For automated design debugging, the performance and functionality of automated debugging tools remain major considerations. This thesis presents a MaxSat formulation for automated debugging to improve performance while allowing engineers to collect temporal information about the nature of the problem. The formulation here also produces a smaller CNF size compared to SAT-based gate-level debugging. However, depending on the circuit and trace, both the solver run time and the number of error locations returned by the debugger can still be prohibitively large. Additional heuristics and better MaxSat solvers are required to further push the performance of contemporary automated debuggers. Mechanisms which can rank error locations according to some heuristics may be helpful to further refine the error locations returned.

Regarding debugger functionalities, automated debuggers also leave plenty of room for improvement. For instance, current debuggers assume that the bug is located within the circuit. However, as assertion languages such as PSL and SVA gain more popularity, errors might occur in the property specifications themselves. Debugging PSL and SVA expressions are another area in debugging that can benefit from further research.

Finally, another relevant area of research are automated correction algorithms. One of the benefits of our MaxSat debugging algorithm is that the solver assigns values to all the variables in the CNF representation of the circuit. The values assigned at the outputs of the MaxSat solutions therefore represent signal assignments that could correct the bug if they were the actual output of the gate. The debugger could suggest fixes to these errors either in the waveform viewer or in the actual RTL code.
Bibliography


