NOVEL DIRECTIONS IN DEBUG AUTOMATION FOR SEQUENTIAL DIGITAL DESIGNS IN A MODERN VERIFICATION ENVIRONMENT

by

Djordje Maksimovic

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 © 2015 by Djordje Maksimovic
Abstract

Novel Directions in Debug Automation for Sequential Digital Designs in a Modern Verification Environment

Djordje Maksimovic

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2015

Digital systems are growing in complexity, introducing significant verification challenges in the design cycle. Verification has grown to take 70% of the design time with debugging being responsible for half of this effort. Automation has mitigated part of the resource-intensive nature of rectifying erroneous designs. Nevertheless, these tools have scalability difficulties and thus, verifying large designs requires further innovation. This thesis introduces three novel methodologies that address this issue. First, a framework to verify designs with multiple clocks using Quantified Boolean Formula satisfiability is presented. It models an iterative logic array representation with universal quantification to achieve synchronization. Next, a framework that ranks revisions based on their likelihood of being responsible for a particular failure is developed. Finally, an automated correction algorithm is discussed. It takes an erroneous design, and corrects failures caused by typical source code mistakes. Experiments demonstrate the benefits of augmenting the verification cycle with these methodologies.
Acknowledgements

First and foremost, I would like to thank my advisor Professor Andreas Veneris for his understanding, advice and his drive for academic research. Without his guidance, I would not have been able to improve my engineering and communication skills to their current state.

I would also like to thank my family, who have been there for me from the beginning. My engineering father, Vladimir Maksimovic, for his aid throughout my life and for the aid with funding my undergraduate education. My mother, Olivera Maksimovic, for always being there for me, and my sister, Katarina Maksimovic, for all the support. I love you all.

These past two years have been some of the hardest in my life, but when I look back on it now, it has been a very intellectually simulating period. I would like to thank my research group for all of their support. John Adler, Ryan Berryhill, Brian Keng, Bao Le, Zissis Poulos, you have all been awesome people to have worked with. I send you all my best regards in your future endeavors, and I hope to work with you again.

I would also like to thank Andrew Becker and Professor Paolo Ienne from the Ecole Polytechnique Federale de Lausanne. It has been a great collaboration working with them.

My sincerest thanks go out to my masters’ defense committee members Professor Jason Anderson, Professor Vaughn Betz and Professor Aleksandar Prodić. Their suggestions have aided in enhancing the quality of my dissertation.

Finally, I would like to express my gratitude to the University of Toronto for their academic resources and their graduate funding. Specifically, I would like to thank them and the respective funding providers for the Bell Scholarship and the Queen Elizabeth II/Montrose Werry Graduate Scholarship in Science and Technology.
## Contents

List of Tables vii  
List of Figures viii  
List of Algorithms x  

1. **Introduction** 1  
   1.1 Background and Motivation 1  
   1.2 Purpose and Scope 4  
   1.3 Thesis Outline 6  

2. **Background** 7  
   2.1 Introduction 7  
   2.2 Boolean Satisfiability 7  
      2.2.1 Conjunctive Normal Form Representation 9  
   2.3 Quantified Boolean Formulas 11  
   2.4 Representing Sequential Circuits 13  
      2.4.1 Single Clock Domain Designs 14  
      2.4.2 Multiple Clock Domain Designs and Mixed Memory Types 14  
   2.5 Functional Verification 17  
      2.5.1 On-line Verification 19  
      2.5.2 Off-line Verification 20  
      2.5.3 SAT-based Automated Debugging 22
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>CNF formulas for elementary gate types</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>Multiple Clock Domain BMC Results</td>
<td>42</td>
</tr>
<tr>
<td>4.1</td>
<td>Benchmark Statistics</td>
<td>63</td>
</tr>
<tr>
<td>4.2</td>
<td>Revision Ranking Performance</td>
<td>67</td>
</tr>
<tr>
<td>5.1</td>
<td>A list of common error library rules</td>
<td>73</td>
</tr>
<tr>
<td>5.2</td>
<td>Syntax-guided Error Correcting Experimental Results</td>
<td>79</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Multi-Designer Verification Cycle ............................................. 2

2.1 Formula decision tree .............................................................. 9
2.2 Combinational logic circuit that is utilized to illustrate property checking ... 11
2.3 The decision tree for the QBF formula in Example 2.2 ......................... 13
2.4 Single clock domain design and its ILA ......................................... 14
2.5 Transformation of multi-clock DFFs to equivalent uni-clock components .... 15
2.6 The ILA representation for a multiple clock domain design ..................... 16
2.7 The on-line verification flow ...................................................... 19
2.8 The off-line verification flow ..................................................... 21
2.9 The potential error locations that can excite failure \( f \) ....................... 23
2.10 A BMC construction ............................................................... 25
2.11 Version Control System structure ............................................ 26

3.1 Circuit diagram of two domain clock synchronization .......................... 31
3.2 Circuit diagram of the inc module ............................................ 32
3.3 Timing diagram of a system with the lowest common multiple gc1k ........... 33
3.4 Circuit diagram of two domain divider-based clock synchronization .......... 37
3.5 Ratio of CNF size to number of cycles ....................................... 39
3.6 Unsigned integer divider representation ..................................... 40
3.7 Ratio of CNF size to bound k for design ac97_ctrl1 (in log scale) .......... 44
3.8 Ratio of solving time to bound k for design vga (in log scale) .............. 45
3.9 Peak memory usage when running each solver

4.1 Flow graph of the proposed revision debug methodology

4.2 Clustering to prioritize error locations

4.3 a) K-Means using a cluster center b) Affinity Propagation clustering

4.4 Converting an area graph to a point graph

4.5 Illustration of Support Vector Machine classification

4.6 Word occurrences in a feature list from eleven OpenCores designs

4.7 Cluster prediction error

4.8 Number of members in each cluster for SDR_CTRL (test 12)

4.9 The classification accuracy for each class with varying cost metrics

5.1 The flow of the proposed automated correction methodology

5.2 A visual example of an error rule

5.3 Constructing a miter with test vectors

5.4 Miter instrumentation that minimizes the number of returned corrections
List of Algorithms

1. Brute Force Approach ................................................. 50
2. Support Vector Machine Training ................................. 59
Chapter 1

Introduction

1.1 Background and Motivation

Modern industrial digital systems have grown in complexity, a reality caused by the necessity for systems capable of performing a multitude of functions. As a result of this, it becomes vital to abstract the design process into various sub-levels to make the process manageable. In the modern design cycle, a high level representation is used to describe the functionality of a design. This representation is typically written in a Hardware Description Language (HDL), such as VHDL [1], Verilog [2], or programmed in the C language [21]. Once a design description is complete, synthesis tools are used to convert the design to the Register Transfer Level (RTL). From this step, further synthesis is required to represent the design at the gate-level, which is performed nowadays with automated tools. Placement and routing tools are used to transfer the design to transistor level, where chip fabrication can finally begin.

At each level, verification is required to ensure that the design abstraction matches the design specifications. At the first level, functional verification is used to ensure that the design functionality correctly represents what is required by the specification. Next, equivalence checking is employed to determine whether the HDL and gate-level representations match. Finally, testing is used to uncover manufacturing defects in the design silicon.

Verifying designs adequately and in a timely manner has become a major challenge in the industrial design process, taking as much as 70% of the design time [24]. It is reported [30]...
that half of this time is spent in debugging, ensuring that functional errors do not escape to
the chip manufacturing stage, where the costs will be steep and may jeopardize the time-to-
market constraints. Despite its importance, various steps of the verification cycle remain a
predominantly manual process.

In the modern design cycle, functional verification can be categorized as either on-line,
where the objective is to expose single failures or off-line, where the design is rigorously tested.
Fig. 1.1 illustrates this cycle.

On-line verification is the task where an engineer analyzes the design to discover an error
trace(s) that exposes a particular failure. An engineer utilizes this verification step to analyze
the correctness of the modules that they are responsible for constructing. On-line verification
has been traditionally performed using simulation tools; however this approach can miss out on hard to detect errors. To aid detection and correction, in the past two decades engineers have augmented the flow with formal tools, which have been shown to improve verification time [40].

Rather than exercising the design functionality in a brute-force manner as in simulation, these tools use mathematical models to represent the complete functionality of a design and the verification task. Following this, decision engines such as Binary Decision Diagrams (BDD), Satisfiability (SAT) and Quantified Boolean Formula (QBF) Satisfiability, are employed to solve the problem. In previous works, tasks such as model checking [14], combinational equivalence checking [27], timing analysis [44], automatic test pattern generation (ATPG) [31], functional debugging [45], etc., have been encoded as SAT problems.

Once on-line verification locates a failure, functional fine-grain debugging is employed to localize the source of the failure. Today, fine-grain debugging has been automated with SAT-based tools, such as those in [45]. They use the error trace(s) to locate possible locations in the design RTL that are responsible for the failure. Next, these locations are mapped back to the HDL representation of the design and they are presented to the engineer. Following this, the engineer must manually locate the actual error location from the list of potential error locations, and correct the failure. Despite of the attractiveness of functional fine-grain debugging, it does not scale well to off-line verification.

Off-line (regression) verification runs extensive test suites, usually overnight, to exercise a majority of the design functionality. Regression verification is usually performed on the entire design to determine if all of the underlying modules correctly function together. When the verification engineer examines the results of regression the next day, debugging is usually performed in a coarse-grain manner by parsing simulation logs and error messages. Candidate errors are discovered and distributed to the appropriate verification or design engineer for a more detailed fine-grain analysis, a process called triage. Following this, the engineers must analyze the data and determine if the error is related to their modules, and correct them if that is the case.

Due to the excessive amount of information that needs to be analyzed after regression, and because engineers working on the same design can be geographically and time-zone dispersed,
it is not a coincidence today that industry declares that for every single designer there are three verification engineers \[30\]. As such, if course-grain debugging is not accurate, this may result in significant delays because the wrong debug case may be passed to the engineer, only for them to examine it and return it back, thus increasing the number of debug iterations. The concepts introduced in this thesis aim to improve the scalability of verification tasks and ease the task of regression verification.

### 1.2 Purpose and Scope

This thesis presents three methodologies that improve on-line and off-line verification. First, a novel verification encoding for multiple clock domain designs is presented. In this formulation, the sequential behavior of the design is explicitly modeled with symbolic universal quantification. The verification problem is encoded in QBF to avoid a memory intensive SAT-based representation. Then, it is used to compute the combinational portion (i.e., transition relation) of each clock only when there is clock activity. Unlike the traditional SAT-based construction, which represents each cycle with copies of the transition relationship, the proposed methodology synchronizes all clocks using a single copy of the transition relation. Furthermore, this formulation removes redundant time-frames that are present in the SAT-based representation from being processed by the QBF solver. Ultimately, this allows a verification task to be computed with less memory when compared to the amount of memory used by SAT-based approaches. The effectiveness of the methodology is illustrated by encoding and solving a bounded model checking (BMC) verification task.

Next, a novel framework that utilizes traditional machine learning techniques along with historical data in version control systems and the results of functional debugging is presented. It is engineered to rank design revisions that may be responsible for a particular failure following regression. The prioritized revisions aid design correction by giving the engineer an intuition of which potential error location is the actual source of error, and when/why the error was introduced. This precise step of analysis aims to ultimately reduce the number of debug iterations. The algorithm applies Support Vector Machine classification to first identify which
revisions are bug fixes and which are incremental improvements. A functional debugging step is performed and followed by Affinity Propagation clustering to rank potential error locations. A weighing scheme is utilized to identify the revisions which are most likely to be the source of the observed failure. Finally, the information gathered is used to direct an engineer’s attention towards revisions which may have caused the failures.

Finally, a syntax-guided re-synthesis tool for source-level error localization and correction is developed. It takes as input a buggy circuit design, at least one failing test vector, and a list of suspect error locations. Using this list and a library of rules characterizing typical source code mistakes, each buggy location is instrumented to be either left unchanged or replaced with a set of possible corrections. Then a QBF problem is solved to determine whether the error can be corrected utilizing these rules and if so, which rule(s) corrects the error. Since all correction rules describe meaningful transformations, the changes presented to the engineer are likely to address the root cause and remove the error. The outcome of this methodology is that it allows typical source code mistakes to be corrected automatically, ultimately reducing the number of debug iterations during on-line verification.

Experiments are conducted on a variety of in-house industrial and OpenCores designs. When the multiple clock domain QBF formulation is compared to traditional SAT approaches, the method achieves memory reductions on average by 76% and in many cases, it is able to complete verification where SAT fails to do so. Experiments demonstrate significant memory savings, with comparable run-times, when contrasted to the state-of-the-art.

The revision debug algorithm assigns a ranking that is on average 68% higher than the ranking achieved by an industrial script-based approach typically used in the industry today. On top of that, it achieves an average run-time overhead of merely 4.631 seconds. Ultimately, this allows actual error locations to be located in a timely fashion, so that the revisions returned to the engineer should give them a sense of what changes could have introduced the observed failures. This allows them to quickly determine appropriate corrections that remove the failure.

Finally, experiments demonstrate that the automated correction methodology corrects 53% of the tests with an average run-time of 70.93 seconds. This proves that many mistakes present in a design are in fact common errors, while the remaining 47% are difficult-to-locate errors.
This allows an engineer to focus their time debugging difficult-to-locate errors rather than spending precious time debugging common errors.

1.3 Thesis Outline

This thesis is organized as follows. Chapter 2 provides the necessary background material for the contributions. Chapter 3 presents the QBF-based encoding for multiple clock domain digital designs. Chapter 4 discusses the clustering-based revision debug algorithm for regression verification. The syntax-guided automated correction methodology is presented in Chapter 5. Finally, Chapter 6 concludes this thesis and discusses research directions for future endeavors.
Chapter 2

Background

2.1 Introduction

This chapter presents material relevant to the contributions presented in this thesis. Section 2.2 gives an overview of Boolean Satisfiability (SAT), while Section 2.3 describes Quantified Boolean Formula (QBF) Satisfiability. Sequential design representations for formal verification are illustrated in Section 2.4. Background information on on-line and off-line verification, bounded model checking, and SAT-based automated debugging is given in Section 2.5. Section 2.6 gives an overview of version control systems. Finally, Section 2.7 summarizes this chapter.

2.2 Boolean Satisfiability

Boolean Satisfiability (SAT) engines are used to determine whether an assignment exists for a boolean formula such that the formula can be evaluated to \texttt{True}. They are typically used in the industry to solve a boolean formula representing a computer aided design (CAD) problem. This is because SAT is an NP-Complete decision problem. This means that all problems in NP can be converted to SAT (\textit{i.e.}, any NP problem can be converted to SAT in a polynomial amount of time and the solution to the SAT problem can be converted into a solution to the NP problem in a polynomial amount of time.) This conversion property of SAT is advantageous because problems in verification, such as functional debugging \[45\], ATPG \[31\], etc., can be solved using an optimized SAT solver, \textit{i.e.}, a separate solver for each problem is not required.
Unfortunately, in the worst case an NP-Complete problem will take an exponential amount of time to solve. Thus, research in this field has been towards improving the formulas representing the problems and the SAT solvers \[6,8,28,34,35\].

A propositional boolean formula \(\Phi\) is constructed from a set of boolean variables \(\{a, b, c, \ldots\}\), where each variable assumes the assignment \textit{False} (denoted by 0), or \textit{True} (denoted by 1). The logic relation that a propositional formula represents is built using boolean connectives, such as \(\neg\) (negation), \(\land\) (logical AND, or conjunction), \(\lor\) (logical OR, or disjunction) and parentheses. Negation takes a boolean formula enclosed by a parenthesis or a variable and inverts its truth value \(i.e.,\) if the formula is \textit{True} (\textit{False}), its negation is \textit{False} (\textit{True}). Let \(A, B\) represent two boolean formulas enclosed by parenthesis or two variables, then logical AND (OR) is written as “\(A \land B\)” (“\(A \lor B\)”) and it evaluates \textit{True} only if \(A\) is \textit{True} and (or) \(B\) is \textit{True}, otherwise it evaluates \textit{False}. With these properties, the SAT decision problem is defined as follows:

**Definition 2.2.1** *Given a propositional formula \(\Phi\), the SAT decision problem is the task of determining whether \(\Phi\) has a model, \(i.e.,\) an assignment to the variables of \(\Phi\) such that \(\Phi\) evaluates to \textit{True}.*

A propositional formula that has a model is said to be satisfiable or \textit{SAT}, otherwise it is unsatisfiable or \textit{UNSAT}. In other words, an \textit{UNSAT} formula evaluates to \textit{False} for every assignment to the variables of \(\Phi\). Also, a SAT solver is a tool that solves the SAT decision problem for a formula \(\Phi\) by determining a satisfying assignment, if one exists (otherwise it returns \textit{UNSAT}).

The following formula will be used as an example for determining whether a formula is \textit{SAT}:

\[
\Phi = (a \lor b \lor c) \land (\neg a \lor b) \land (c)
\] (2.1)

This formula is \textit{SAT}, because it has a model \(\{a = \text{True}, b = \text{True}, c = \text{True}\}\) which evaluates \(\Phi\) to \textit{True}. A decision tree is utilized to illustrate how this formula is determined to be \textit{SAT}. The decision tree is a binary tree of depth \(n + 1\), where \(n\) is the number of variables, that illustrates the evaluation of \(\Phi\). At each level of the tree, the internal nodes are labeled as a unique variable from \(\Phi\). The left (right) branch is labeled 0 (1), which represents an assignment of \textit{False} (\textit{True}) to the variable of the respective node. The leaves of the tree represent the
evaluation of $\Phi$ after each variable is assigned according to the assignments made by traversing the path from the root node to the leaf. Fig. 2.1 illustrates this process.

Similarly to NP problems, some PSPACE problems, such as model checking [14], can also be solved by a SAT solver since they can also be represented as a boolean formula. In terms of complexity, these problems are harder to solve than NP problems; however in practice some of these problems can be solved in a timely manner due to the optimizations made in modern SAT solvers.

To solve a verification task in SAT, a polynomial algorithm that can convert a digital circuit diagram to a boolean formula is required. This algorithm is the Tseitin transformation [47] that can represent a combinational logic circuit as a boolean formula in conjunctive normal form.

### 2.2.1 Conjunctive Normal Form Representation

A formula in Conjunctive Normal Form (CNF) is a boolean formula constructed from a conjunction (logical AND) of clauses. A clause is formed through the disjunction (logical OR) of a set of literals. A literal is a variable or its negation. An example of a CNF formula is given by:

$$\Phi = (\neg a \lor f) \land (\neg h \lor f) \land (a \lor h \lor \neg f) \land (b \lor \neg h) \land (c \lor \neg h) \land (\neg b \lor \neg c \lor h)$$  \hspace{1cm} (2.2)

which is the CNF encoding of the circuit in Fig. 2.2. For a formula $\Phi$ to be satisfiable, each clause must itself be satisfiable (i.e., evaluate to True).
Table 2.1: CNF formulas for elementary gate types

<table>
<thead>
<tr>
<th>Gate</th>
<th>Function</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>$y = x_1 \land x_2 \land \cdots \land x_n$</td>
<td>$\bigwedge_{i=1}^{n} (x_i \lor \neg y) \land \left( \bigvee_{i=1}^{n} \neg x_i \lor y \right)$</td>
</tr>
<tr>
<td>OR</td>
<td>$y = x_1 \lor x_2 \lor \cdots \lor x_n$</td>
<td>$\bigwedge_{i=1}^{n} (\neg x_i \lor y) \land \left( \bigvee_{i=1}^{n} x_i \lor \neg y \right)$</td>
</tr>
<tr>
<td>NOT</td>
<td>$y = \neg x_1$</td>
<td>$(x_1 \lor y) \land (\neg x_1 \lor \neg y)$</td>
</tr>
<tr>
<td>BUFFER</td>
<td>$y = x_1$</td>
<td>$(x_1 \lor \neg y) \land (\neg x_1 \lor y)$</td>
</tr>
</tbody>
</table>

The CNF representation is used because any combinational circuit can easily be converted into that representation by the process of the Tseitin transformation. Gate signals are converted to variables and the gate function is represented by a CNF formula utilizing these variables. For each of the elementary gates (AND, OR, NOT, BUFFER) there is an associated CNF representation of the gate, which can support any number of inputs. Table 2.1 displays these representations, where the gate output is $y$ and gate inputs are represented by $x_1, x_2, \ldots, x_n$. To generate a CNF formula of the combinational logic circuit, an engineer must simply construct a CNF formula that is the conjunction of the clauses that represent every gate in the circuit. Therefore, this transformation is of $O(n)$ time, where $n$ is the number of gates in the circuit.

When constructing a CAD problem in CNF the CAD task itself must be converted to CNF. This typically consists of representing the task as a circuit and then converting it to CNF. Any constraints, such as input/output constraints, that the problem includes can easily be converted to CNF by adding a unit literal clause (i.e., a clause with a single literal) that forces the assignment of a variable during the solving process.
Example 2.1 Fig. 2.2 illustrates a combinational circuit. A property checking verification task is performed to determine whether the circuit can output a logic 0 when the inputs \( b \) and \( c \) are set to a logic 1. The CNF formula for this task is:

\[
\Phi = (\neg a \lor f) \land (\neg h \lor f) \land (a \lor h \lor \neg f) \land (b \lor \neg h) \land (c \lor \neg h) \land (\neg b \lor \neg c \lor h) \land (b) \land (c) \land (\neg f)
\] (2.3)

In this formula, the first three clauses represent the OR gate and the next three represent the AND gate. Notice that the last three clauses are not related to the CNF formulas of the gates. These are added to encode the property checking task. For the property to hold, the formula must be satisfiable. Thus, \( b \) and \( c \) must be set to a logic 1 and \( f \) must be set to a logic 0 for the final three clauses to be satisfiable. However, by performing this action the second clause evaluates to \( \neg h \), while the sixth clause evaluates to \( h \). Since \( h \) can only be either True or False, then one of the clauses must evaluate False for every assignment to the variables. This indicates that \( \Phi \) is UNSAT and that the property does not hold. By observing Fig. 2.2 it can be seen that indeed the property does not hold because an AND gate outputs a logic 1 when its inputs are a logic 1, and an input of logic 1 is a controlling signal for the OR gate, which forces its output to be a logic 1.

2.3 Quantified Boolean Formulas

Many CAD problems exist that belong to a higher complexity hierarchy, for example model checking in PSPACE. Although these problems can be solved with a SAT solver, a dedicated solver for a particular complexity hierarchy may solve the problems with less memory and/or time requirements. For those problems it is better to represent them using another theory.
Quantified Boolean Formula (QBF) satisfiability is a natural extension to SAT. QBF is a PSPACE-Complete problem and thus, is in a higher complexity class than SAT. While many problems in verification can be converted to SAT, a portion of these problems, such as model checking, are PSPACE problems which are more easily represented in QBF.

The QBF theory is comparable to SAT, except that variables can be quantified as either existential ($\exists$) or universal ($\forall$). Given a formula $\Phi$ which has the same definition as the SAT counterpart, the existential variables of $\Phi$ have the same definition as the variables for SAT (i.e., for the QBF formula to be SAT, the evaluation of $\Phi$ must lead to True for an assignment to the existential variables.) On the other hand, for universal variables $\Phi$ must evaluate to True for all assignments (True and False) to the universal variables. A QBF instance in prenex normal form is written as follows:

$$Q_1V_1Q_2V_2\ldots Q_nV_n|\Phi$$

where $Q_i \in \{ \exists, \forall \}$ is a scope (quantification), $Q_i \neq Q_{i+1}$, $V_i$ is a set of Boolean variables, and $\Phi$ is a logic formula in conjunctive normal form (CNF). In this notation, the leftmost (rightmost) $Q_i$ is called the outer (inner) scope, and variables are assigned values moving from the outer scope to the inner one. Also, the variables $V_i$ are given the quantification $Q_i$.

Since QBF is an extension to SAT, any propositional formula can be represented in QBF. For example, Equation [2.3] in Section 2.2.1 can be written in QBF as:

$$\exists\{a, b, c, f\}|(\neg a \lor f) \land (\neg h \lor f) \land (a \lor h \lor \neg f)$$

$$\land (b \lor \neg h) \land (c \lor \neg h) \land (\neg b \lor \neg c \lor h) \land (b) \land (c) \land (\neg f)$$

Note, when solving a QBF formula, the quantification order matters. For example, $\exists a \forall b | \Phi$ is not the same as $\forall b \exists a | \Phi$. The solving process of a QBF formula can also be visualized with a decision tree; however the variables assigned at each level must follow the quantification order given for the problem. Furthermore, $\Phi$ must evaluate to True for paths along both branches of any universal variable to satisfy the QBF. If $\Phi$ is SAT for one assignment to a universal variable, but is UNSAT for the other assignment, then the QBF is said to be UNSAT.
Chapter 2. Background

Example 2.2 Suppose the following QBF formula in prenex normal form must be solved:

$$\exists a \forall b \exists c (a \lor b \lor c) \land (\neg a \lor b) \land (c) \quad (2.6)$$

The decision tree of Fig. 2.3 shows how this formula evaluates to SAT. In the figure, a double lined branch indicates a variable with universal quantification, while a single line branch indicates an existential variable. The dashed paths indicate the assignments that lead to the formula evaluating True. Since two dashed paths traverse across both branches of the only universally quantified variable and lead to a True evaluation, then this indicates that the QBF formula is satisfiable.

2.4 Representing Sequential Circuits

Almost all designs nowadays are sequential designs. Sequential designs contain memory elements such as D flip-flops (DFF) and latches. Each memory element has an associated input, output and a clock, which is a signal that indicates when state changes should occur i.e., when the memory element records a new value. A state change occurs on the positive edge of the clock for DFFs. On the other-hand, state changes for latches occur when the clock takes the value 1 and the input toggles.

In Section 2.2.1 the CNF formula is discussed and formulas are given for several elementary gates. That representation works well for verification problems with combinational logic designs;
however challenges appear when memory elements are to be converted to CNF for solving verification tasks with SAT engines. The issue that arises with memory elements is that they can change their assignment during the toggling of the clock; however in SAT there is no concept of time and variables can only take one assignment. Therefore, time must be modelled in SAT to solve problems with sequential designs, and the models vary depending on whether the design has one clock domain, many clock domains or mixes memory element types.

2.4.1 Single Clock Domain Designs

In the simplest case of one clock domain, time can be modelled by “unrolling” the design to its iterative logic array (ILA) representation [33]. This involves duplicating the combinational part (transition relation) of a sequential circuit into \( k \) time-frames, such that the next-state variables of the current time-frame are connected to the current-state variables of the next time-frame. Fig. 2.4(a) displays a sequential circuit, while Fig. 2.4(b) illustrates the corresponding ILA for three cycles, where \( a, b, c (a^i, b^i, c^i) \) are input (at cycle \( i \)), and \( x, y, z (x^i, y^i, z^i) \) are output (at cycle \( i \)). Note that the number of cycles which can be represented by an ILA is equal to the number of time-frames.

2.4.2 Multiple Clock Domain Designs and Mixed Memory Types

If a design has multiple clock domains or includes latches, then representing the design in SAT is a challenge because each state element changes states at a different time. State elements,
such as DFFs, update their values at a specific frequency while others, such as latches, can update any time its clock is active. Furthermore, SAT has no direct representation of time and as such cannot directly model state elements.

This challenge is solved by the methodology in [26], where a copy of the transition relation is created whenever there is a state transition in a clock domain. The ingenuity in that work is to unify the clock domains by replacing state elements with components using the same clock, similar to those shown in Fig. 2.5, which behave in the same manner as the original state elements. Each new component contains DFFs that make use of a single global clock $gclk$ where the positive edge of this clock indicates a state transition in at least one of the original clock domains. Thus, the resulting circuit after replacement is a single clock domain circuit that behaves in the same manner as the original circuit. The difference now is that the new circuit can be modelled in SAT by converting it to an ILA.

In more detail, Fig. 2.5 displays the transformation for components replacing multi-clock DFFs. In that figure, an edge detector, denoted by the dashed box, is utilized to detect an edge of the original clock. When this occurs, a flip-flop clocked by a global clock takes the new value, otherwise the previous value is held at the output. By replacing all DFFs with this circuitry, a single clock domain design is generated that retains the functionality of the original design.

A similar process can be applied to latches and other memory types. In the case of latches,
the same process used for flip-flops can be used, except that a clock signal level detector is utilized, rather than a positive edge detector, to trigger latch updating. After unifying the domains, the ILA representation is generated.

Fig. 2.6 displays the ILA representation for a sequential logic circuit with multiple clock domains, where \( a, b, c \) \((a_i, b_i, c_i)\) are input (at time-frame \( i \)), and \( x, y, z \) \((x_i, y_i, z_i)\) are output (at time-frame \( i \)). Also, \( c_i, 1 \leq i \leq n \), corresponds to the \( i \)-th distinct clock domain. For each domain \( c_i \), \( s_{c_i}^0, \ldots, s_{c_i}^{m_{c_i}} \) are used to represent state element values at cycle 0, \ldots, \( m_{c_i} \) respectively, where \( m_{c_i} \) is the maximum cycle that clock \( c_i \) can reach. Finally, \( t_g^i \) is denoted as the \( i \)-th cycle of \( gclk \), \( i = 0, 1, 2, \ldots, m_g \) where \( m_g \) is the maximum cycle \( gclk \) can reach.

Due to the process of unifying the clock domains and the generation of a global clock, redundant time-frames where no state changes occur can be generated in the ILA. Redundant time-frames pose bottlenecks in traditional SAT instances with multiple clock domains as they require extra time and memory to compute. Two types of redundant time-frames exist: extra frames, and waiting frames.

Extra frames are instantiated by the global clock due to state changes in other clock domains. On a positive edge of the global clock, there is no way to distinguish whether an edge occurs in two or more clock domains. All that is indicated is that at least one clock domain is transitioning states. To ensure correct functionality of the design, a new frame must be created for every
clock domain. For clock domains that are not transitioning states, this results in duplicate computations of their next state assignment. For example, if clock domain \( c_1 \) is two times faster than \( c_2 \), then in the ILA representation of Fig. 2.6, the transition relation of \( c_2 \) during \( t^1_g \) is an extra frame as it corresponds to the second half of \( c_2 \)'s period. The second clock domain would not be on a positive edge until \( gclk \) reached \( t^2_g \).

On the other hand, waiting frames arise when one clock domain is waiting for an acknowledge from another clock domain. Assuming inputs are held constant, suppose that in Fig. 2.6, domain \( c_1 \) is waiting for data from \( c_2 \), this means that during \( t^0_g \) and \( t^1_g \), \( c_1 \) may have had no state changes, because it is waiting for an acknowledge from \( c_2 \) that would appear on \( t^2_g \). Consequently, any assignment to the transition relation during \( t^0_g \) is redundant because it is identical to the assignment during \( t^1_g \). Waiting frames are not generated by clock unification, but may appear in the original multiple domain design.

### 2.5 Functional Verification

Functional verification is the task where a design is tested to determine whether it conforms to the design specifications. A sub task of verification called functional debugging aims to correct any failures that are identified. Errors in the design may stem from multiple sources such as timing constraint violations; however in this thesis errors at the RTL are addressed.

Functional verification is traditionally performed using simulation tools that exercise the design functionality. These tools produce the output that the design would present in normal operation. After simulation, the primary output must be matched with expected results from a checker to discover failures. Apart from traditional simulation, assertions are also used to determine when a failure occurs. During simulation they combine signal checking to monitor signal mismatches and record a failure as it happens. In the modern verification cycle, formal tools are employed to improve the quality of verification. They use mathematical models to exercise hard to detect errors. Tasks such as bounded model checking \([10]\), assertion generation \([48]\), unbounded model checking \([10]\), etc. have been performed in this manner and have proven to improve verification quality and decrease the amount of design time spent in verification.
In functional verification there are two distinct concepts which will be discussed in this section, errors and failures.

**Definition 2.5.1** An error is a design component (gate, wire, input, output, etc.) in the RTL that, for an input vector, produces a functional mismatch at a primary output with expected values from a checker or a correct (golden) model.

An error location is an erroneous design component. In this thesis, the terms “error location” and “error” are used interchangeably.

**Definition 2.5.2** A failure is a discrepancy at a primary output or any observation point between the observed values and expected values from a checker or a golden model for an input vector.

In other words, a failure is caused by an error in the design RTL. In this thesis it is assumed that each failure has an associated error trace, which is a set of signal assignments that describe the conditions in which the failure can be reproduced. By simulating the erroneous design with the error trace, an engineer can observe the same mismatch between expected and observed values. Unfortunately, failures without associated error traces do exist. Problems such as state unreachability are difficult to debug, because states that should be reachable can be determined to be unreachable through model checking. The result of model checking will indicate that the state is unreachable because a property is violated; however an input vector(s) exhibiting that the state cannot be reached does not exist.

Errors in a design can manifest themselves as problems in the design implementation, an incorrectly implemented test-bench, or in the verification environment, for example an incorrectly written assertion. Problems in the design implementation are called design errors and are caused by a change in the specification, the human factor, or errors in the design tools. On the other hand, the later two are known as verification errors and are caused by an incorrect translation of the design specification to verification code.

Functional verification can be abstracted into two tasks depending on the complexity of the design to be implemented. The first task, on-line verification, is performed by a single engineer
to verify design components that they are responsible to construct. The second task, off-line verification, is employed to verify large designs that were built by multiple engineers, possibly in geographically dispersed regions.

2.5.1 On-line Verification

On-line verification is performed by a single engineer to discover errors in their own designs and it is illustrated in Fig. 2.7. The engineer will utilize simulations or formal verification tools to expose a set of failures. If no failures are detected, then the design is verified and more strenuous verification, such as off-line verification, begins. On the other hand, if failures are detected, then the associated error trace is used to debug the design and identify the error locations that are responsible for the failures. This can either be performed using automated tools or manually. Finally, once the error locations are identified, the design component must be corrected. Design correction can similarly be performed manually or with automated tools.
The primary focus of on-line verification is to either verify a module of a larger design, or to test the functionality of a complete design. In either case, the basic functionality of the design is tested. This includes simulating the design or module under typical use scenarios and ensuring that important properties are met. At this stage, rigorously testing the design for security issues, unexpected behavior, or other unusual states would be time consuming. Consequently, this testing is deferred to off-line verification.

2.5.2 Off-line Verification

Off-line (regression) verification is typically performed overnight to rigorously test a large design that was constructed from the labor of multiple engineers. In this task, verification engineers will produce test-benches that exercise the design functionality according to the most up-to-date design specification. The verification engineers may also provide negative tests, such as invalid input, that are meant to determine whether the design fails gracefully or handles the failure appropriately. The primary concern is to identify any error that the design engineers may have overlooked. Fig. 2.8 displays the off-line verification cycle.

In the first phase of regression verification, a complete (but possibly erroneous) design is built from the modules that each design engineer is responsible for constructing. Following this, a simulation is performed using an extensive test-suite. If no failures are detected, and the test-suite is complete and not erroneous itself, then this indicates that the design phase has been completed. However, when the design is erroneous, the number of failures returned by the simulation is typically extensive.

At this phase, the design must go through a task called coarse-grain debugging. This involves parsing simulation logs and distributing failures to verification and design engineers who are best suited to correct the failures. This difficult process remains a predominantly manual one for two major reasons. First, since automated debugging is an NP-Complete problem, applying this process to every single failure will take a protracted amount of time. Second, failures identified by this step can appear different, but may have come from the same error location. For example, an error location may have triggered an assertion and the same error may have caused a mismatch at a primary output between the observed values and the values
expected by a checker. Clearly, correcting the error location for one failure can also correct these failures.

Automated tools at the coarse-grain debugging phase have been targeted towards a task called failure triage \[38\]. This task aims to determine similarities between failures returned by verification and organize them into bins. For each bin there is an associated engineer who is the best candidate for handling the failures that are grouped into the bin(s). Failure binning has proved successful in practice, because if failures originating from the same error location are given to the engineer who introduced the error, then they may have insight into which

Figure 2.8: The off-line verification flow
failures are more useful than others. Without failure binning, failures may be sent to the wrong engineers who will spend time on them only to return them back to the verification engineer unresolved, which will increase the number of debug iterations.

In the final phase after failure triage, each engineer debugs their design modules using failures in their bin. The same debugging methods used in on-line verification can be employed at this time. In this phase, failures are either corrected or they are returned to the verification engineer. Once all failures have been corrected, the off-line verification cycle repeats until the design is proven error-free.

2.5.3 SAT-based Automated Debugging

Functional debugging is the process of determining the root cause of a failure and has been shown to be a NP-Complete problem \[45\]. This process can be performed in multiple ways such as through the use of simulation, BDDs, SAT or manual methods. However, SAT has recently proven \[45\] to be the most efficient method for automated debugging. SAT-based automated debugging is the process of locating design components in the design RTL that are responsible for the failures using SAT decision engines. The error can be any component (e.g. gates, RTL source code, design modules) depending on the implementation of the design.

In automated debugging, a potential error location is an RTL component which when modified can correct the observed failure. A particular failure \(f_i\) may have multiple potential error locations, denoted as \(E_i = \{e_i^1, e_i^2, \ldots, e_i^n\}\), where \(e_i^j\) is a single potential error location in Fig. 2.9. Intuitively, this occurs because there is at least one component which is the actual error location, and other components which, when corrected, can mask the failure. Furthermore, multiple different failures may be the result of the same error. Suppose in Fig. 2.9 there is another failure at \(2t\) cycles. If the design returned to the initial state in cycle \(t + 1\), and the same input vectors used in the first \(t\) cycles are used for the next \(t\) cycles, then the failure that occurs at \(2t\) cycles is equivalent to \(f_i\) at cycle \(t\) and is caused by the same error.

Prior to debugging, the verification tool takes as input the design under verification (DUV) as well as the high level specification. If a failure is identified, then an error trace is generated.
Following this, the debugging process takes the error trace as well as the DUV and returns potential error locations.

To construct a SAT instance of the debugging process, the transition relation of the design is first enhanced with a set of suspect locations. A suspect location is a design component in the RTL that is either left unchanged, or is allowed to be set to any boolean value. The effect of this is to determine whether changing a signal value at the suspect location will correct the failure observed at the outputs. This construct can be implemented as a hardware multiplexer or an if-then-else construct, or it can be directly coded into CNF. A suspect variable is used to select between leaving the functionality unchanged, or allowing any boolean value.

Next, the ILA of the transition relation is constructed. Suspect variables are not replicated since they represent the same location regardless of time-frame. The error trace predicates are applied to the inputs of the ILA, and the expected response is constrained to the output of the ILA. Finally, a cardinality constraint is applied to the formula, which controls the number of simultaneous active suspect variables. This construct is then converted to CNF and its encoding is given by the following SAT formula:

$$Debug^p_k = S^p(s^p) \land \phi_N(E_i) \land X(x) \land Y(y) \land Ten^k_{ILA}(s^p, x, y, E_i)$$

(2.7)

where $S^p$ is the initial state constraint for time-frame $p$, $X$ ($Y$) is the input (output) set constraint, $Ten^k_{ILA}$ is the ILA of the enhanced transition relation, which is unrolled for $k$ time-frames and $\phi_N(E_i)$ sets the cardinality constraint for a set of suspect variables (potential error locations) $E_i$ belonging to a failure $f_i$.

When this formula is SAT, the active suspect variables (assigned to True) indicate the
suspect locations which are possible error locations. If the formula is \textit{UNSAT} then the suspect location cannot be corrected to fix the error. To find all possible errors for a set of suspect variables, a blocking clause is added to the debugging instance which will block the active suspect variables from appearing again as a satisfying assignment. Then the debugging problem is solved iteratively by adding new blocking clauses and sending the instance to the solver. Eventually, when the solver returns \textit{UNSAT}, then all possible errors have been found.

\subsection{Bounded Model Checking}

Bounded Model Checking [5, 9, 20] (BMC) is the task of determining whether a design implements a set of properties within a specific bound of \( k \) cycles. BMC has been proven time and memory efficient [20] when finding failures quickly, but cannot prove that errors do not exist. This is true, because an error may not be excitable for a particular constraint or the bound \( k \) may not be large enough to excite and propagate the error signal. BMC is typically used to detect false properties (\textit{i.e.}, properties that should not occur in a correct operation.) This is useful when implementing BMC as a SAT instance, as a satisfiable instance will produce a complete error trace (all internal signal values) which caused the failure. This can provide insight into identifying and correcting hard to detect failures \textit{i.e.}, failures that occur when a design is operated in atypical conditions.

To construct a BMC instance in SAT, as illustrated in Fig. 2.10, properties are typically synthesized to a RTL representation denoted as \( T_{prop} \). Following this, the design ILA is constructed for \( k \) cycles. Initial states are constrained and signals from each time-frame that are to be monitored for a property violation are connected to \( T_{prop} \). Finally, this is encoded into CNF and solved by a SAT engine. A SAT result indicates that the property is violated, and an error trace (assignments to all variables) is returned. On the other hand, an UNSAT result indicates that the property has not been violated.
2.6 Version Control Systems

A version control system (repository) is a system of programs that manage changes to documents, code, or other types of information. In verification, these systems indicate what type of changes were implemented at a particular stage of the design process after the last successful verification step. Revisions store the changes that have been made by an engineer, while committing is the act of taking a temporary change and converting it to a revision in the version control system.

When groups of designers utilize version control systems, each designer has their own local copy of the entire, or part of, the design stored in the repository. Each designer is free to make changes to their own local copy. When a designer is ready to make their changes permanent they must commit their local copy to the repository. The remaining engineers will then update their local copy to include the new changes. A conflict occurs when multiple designers modify the same location in a file, and the version control system does not know which change should take precedence over the other. In these cases merging must be performed, by combining the changes manually or selecting one of the conflicting changes and discarding the rest. Engineering teams may require that any commits be bug-free, therefore on-line verification is typically performed by an engineer prior to committing a revision. Off-line verification is performed after a milestone is reached on the entire design stored in the repository. Fig. 2.11 illustrates the structure of a version control system. As could be seen, the steps of code modification, committing, merging and updating are performed iteratively towards the completion of the project.
Chapter 2. Background

There is a multitude of data stored by version control systems that proves to be useful during functional verification. This data includes the author of the revision, the date of the commit, a commit log that describes the changes an engineer has made and the changes themselves. Proper use of this data can aid in improving verification time and quality.

2.7 Summary

In this section, background material for the contributions of this thesis were presented. The topic of Boolean Satisfiability was introduced, followed by a discussion on QBF Satisfiability. Next, the ILA was introduced for sequential design representations. Finally, functional verification concepts were illustrated and an overview of version control systems was given.
Chapter 3

Clock Synchronization in a QBF-based Verification Flow

3.1 Introduction

To reduce power consumption, modern designs are architected with multiple clock domains, a practice that has introduced new challenges during verification [10]. To overcome this challenge, recent research has proposed a clock domain unification scheme that generates a single clock domain [26]. This technique allows one to verify the design without having to worry explicitly about the synchronization of different domains. While the unification technique overcomes the synchronization issue it can introduce redundant time-frames. In essence, these redundant frames are copies of the transition relation where no state change occurs. As a result, additional time and memory are devoted to verify these unnecessary time-frames.

In more detail, the work of [26] achieves synchronization by converting state elements into functionally equivalent components clocked by a global clock. Conceptually, the global clock acts as a data probe where its positive edge occurs during a state transition of a clock domain in the design. This results in a single clock domain circuit, clocked by the global clock, that behaves identically to the original multiple clock domain circuit. Following this, an iterative logic array (ILA) [33] is generated, an action also knows as “circuit unrolling”. However, in the presence of multiple clocks, one clock may have to wait for computation from another clock. This wait time
will manifest itself as a waiting frame(s), where state changes do not occur. As mentioned in the previous section, extra frames are instantiated by the global clock due to state changes in other clock domains. Collectively, waiting frames and extra frames are called redundant time-frames. With large circuits, the memory capacity becomes a concern because ILA memory size scales with the number of time-frames. Over many cycles, the memory requirements may become substantial \[33]. This realization prompts into developing memory-efficient methodologies.

In this chapter, a novel verification methodology for multiple clock domain designs, that tackles key inherent challenges of the problem, is presented. In an introductory formulation, the ILA representation is explicitly modeled with symbolic universal quantification. This is employed to compute the transition relation of each clock only when there is clock activity. Later, the original formulation is extended with the use of a divider circuit that saves on time and memory when compared to the original one. An additional contribution of the work is that it presents these two new verification schemes in the context of Quantified Boolean Formula (QBF) satisfiability. By using QBF, it avoids the memory intensive ILA representation.

Experiments on BMC demonstrate the efficiency of the proposed QBF-based multiple clock verification framework. Specifically, when compared to traditional SAT approaches, the method achieves memory reduction on average by 76\% and in many cases, it is able to complete verification where SAT fails to do so. Evidently, advances in QBF solvers, a fast growing field, will further improve the runtime of the proposed methodology.

The remainder of this chapter is outlined as follows. Section 3.2 contains background material including prior art. Section 3.3 describes the basic clock domain synchronization approach while Section 3.4 extends it with the use of a divider to synchronize the different clocks. Section 3.5 presents the experiments and Section 3.6 concludes this chapter.

### 3.2 Preliminaries

#### 3.2.1 Prior Art

In \[22\], the authors present a methodology for modelling the relations between clock domains, such that designs can be verified under any formal technique. This is done by translating
Chapter 3. Clock Synchronization in a QBF-based Verification Flow

Clock constraints into state machines representing these relations. In the methodology, the relationship between clock domains is encoded using clock constraints specified by the engineer. The work of [12] presents a method to abstract phase relations between clock domains. It assumes that clocks are generated within the netlist of the design and the first task is to find the clocks by matching nets with repeating bit patterns. They employ heuristics to decide on the number of phases they wish the design to have, and use this information to reduce the number of state variables. The proposed methodology assumes that clocks are generated from the same source and phase differences are negligible.

Of more interest to our work is the methodology in [26] that unifies clock domains with the use of a single global clock. The authors use embedded flip-flops clocked by a global clock, which oscillate at a frequency where the positive edge indicates a state change in a clock domain. Since the number of modelled cycles in SAT or QBF is always finite, a global clock that behaves in this manner can always be found. A positive edge detector is used to detect an edge of the original clock. When this occurs the flip-flops take the new value, otherwise the previous value is held at the output. By replacing all the state elements with this circuitry, a single clock domain design is generated that retains the functionality of the original design. After unifying the domains, the ILA representation is generated.

3.2.2 Notation

The following notation is used throughout this chapter. Assume a design with \( n \) clock domains, where \( c_i, 1 \leq i \leq n \), corresponds to the \( i \)-th distinct clock domain. For each domain \( c_i \), \( s_{c_i}^0, \ldots, s_{c_i}^{m_{c_i}} \) is used to represent state element values at cycle 0, \( \ldots, m_{c_i} \) respectively, where \( m_{c_i} \) is the maximum cycle that is modelled for clock \( c_i \). It is assumed that every state element is always clocked by the same clock domain. Predicate \( B(s_{c_i}^k) \) is the set of safety properties which should not be violated in \( k \) cycles of \( c_i \) and is used to construct the BMC instance. It is assumed that all clock domains have a fixed frequency, they all start at the positive edge of cycle 0, and state changes occur at the positive edge of the clock. Similar to the work in [26], a global clock \( gclk \) is generated with a frequency that is the lowest common multiple of all the original \( n \) clock domains. For example, if there exist two clock domains of 2 and 3 MHz each,
Chapter 3. Clock Synchronization in a QBF-based Verification Flow

the global clock is at 6 MHz. In the case where two or more clocks are truly unsynchronized (i.e., their frequencies do not have a lowest common multiple), then the frequency of the global clock must be approximated. In this case, the property where its positive edge occurs during a state transition of some clock domain is only true for a finite number of modelled cycles. For example, if there exist two clock domains of 1.333... and 2 MHz each, then the clocks are simulated for the number of cycles that are to be modelled, and a global clock frequency is selected such that every positive edge of the two clocks lines up with a positive edge of the global clock. Notation \( t_i^g \) is denoted as the \( i \)-th cycle of the global clock, \( i = 0, 1, 2, \ldots, m_g \) where \( m_g \) is the final cycle of \( gclk \) that is modelled. For time-frames where the positive edge of \( t_i^g \) does not indicate any state transition, those time-frames are simply not modelled in the formulation. Thus, the positive edge of those \( t_i^g \) that are modelled indicate a state transition in at least one of the original clock domains. Finally, \( t_i^{c_i} \) is the cycle number of the global clock \( gclk \) when clock domain \( c_i \) is at the positive edge of its cycle \( j \).

3.3 QBF-based Clock Synchronization

In this section a novel clock synchronization formulation is presented that eliminates redundant time-frames. This is possible by tailoring the solution around a QBF implementation. For the sake of simplicity, the presentation is built for a two clock domain design. This can be easily generalized for a variable number of clock domains.

The framework is depicted in Fig. 3.1. Inside the dashed boxes, a QBF-based ILA encoding is presented \[33\]. Combinational logic clocked by clock \( i \) is represented by the transition relation \( T_{c_i}(s_{c_i}, s'_{c_i}, \text{input}_{c_i}, \text{output}_{c_i}, CDC_{c_i,c_j}) \), where \( s_{c_i} \in \{ s_{c_i}^0, s_{c_i}^1, s_{c_i}^2, \ldots, s_{c_i}^{m_{c_i}} \} \) \( (s'_{c_i} \in \{ s_{c_i}^1, s_{c_i}^2, \ldots, s_{c_i}^{m_{c_i}} \}) \) is the current (next) state assignment, \( CDC_{c_i,c_j} \) is a set of clock domain crossing signals between \( c_i \) and \( c_j \), and \( \text{input}_{c_i} \) \( (\text{output}_{c_i}) \) is the set of primary input \( (\text{output}) \). The two multiplexers around \( T_{c_i} \) are used to connect the next state assignment of the current cycle to the current state assignment of the next cycle through the use of the select line \( cycle_{c_i} \). In this sense, the value of the select line determines the state of a clock domain at a particular cycle and allows one to model the sequential behavior of the design \[33\].
For the design in the dashed boxes to exhibit correct functionality, signals $cycle_{c_1}$ and $cycle_{c_2}$ should be assigned to cycles that synchronize the two clock domains. The synchronization logic is denoted by the $inc_{c_i}$ hardware module. The purpose of this module is to transition a clock domain to the next cycle when the clock is on a positive edge. If $c_i$ is not on a positive edge then it remains in its current cycle. Its architecture is found in Fig. 3.2. The signal $edge_{j_i}^c$, for all $0 \leq j \leq m_g$, in Fig. 3.2 defines how the clock domain changes cycles, and is the main factor in achieving synchronization. It is a binary signal that must be constrained to one on a positive edge of the clock domain, otherwise it is zero. The purpose of $gclk$ is to order the positive edges of each clock domain temporally, according to the specification.

Returning to the description of Fig. 3.1 the two outer multiplexers essentially encode the current to next cycle assignment for $inc_{c_i}$. In this case $cycle_{c_i} \in \{cycle_{c_1}^0, \ldots, cycle_{c_i}^{m_g-1}\}$ is the current cycle assignment whereas $cycle_{c_i}' \in \{cycle_{c_1}^1, \ldots, cycle_{c_i}^{m_g}\}$ represents the next cycle assignment. These multiplexers are used to model the sequential behavior of the $inc_{c_i}$ module.
Chapter 3. Clock Synchronization in a QBF-based Verification Flow

Notice that when \( \text{cycle}_{c_i}' = \text{cycle}_{c_i} \), this indicates a cycle of the global clock where clock domain \( c_i \) is not transitioning states. The combination of the outer multiplexers and the \( \text{inc}_{c_i} \) module is hereafter called the \textit{synchronizer}.

Clock synchronization ensures that state changes in each clock domain are ordered temporally according to the design specification. In more detail, it ensures that each clock domain is synchronized in terms of valid cycles with respect to the global clock, where a valid cycle is defined as follows:

\begin{definition}
Cycle \( i \) for clock domain \( c_l \) is called valid w.r.t. cycle \( j \) of the global clock \( gclk \) iff \( t_{c_l}^i \leq t_{g}^j < t_{c_l}^{i+1} \), where \( 0 \leq i \leq m_{c_l} \), \( 0 \leq j \leq m_g \), and \( 1 \leq l \leq n \) where \( n \) is the number of clock domains and \( m_{c_l} (m_g) \) is the maximum cycle that is modelled for clock domain \( c_l \) (gclk).
\end{definition}

To achieve synchronization, instances must be prevented where one clock domain is processing a previous cycle and another is processing a future cycle w.r.t. the current cycle of the global clock. It is also necessary to prevent clocks from exercising a different frequency to the one defined by the specification.

\begin{example}
Suppose that in Fig. 3.2 the global clock is on \( t_g^0 \). This means that \( t_{c_1}^0 \) and \( t_{c_2}^0 \) are valid cycles because their edge occurs with \( t_g^0 \). Likewise, if the global clock is on \( t_g^1 \) then \( t_{c_1}^1 \) and \( t_{c_2}^1 \) are valid, but \( t_{c_1}^0 \) is not, as \( c_1 \) has temporally completed that cycle.
\end{example}

In the following, it is shown that the synchronizer hardware does indeed synchronize the clock domains of the design.
Theorem 3.3.2 The synchronizer generates only valid cycles.

Proof: To prove the theorem, induction is applied on the number of cycles for the global clock. For the basis, let the global clock be on $t^0_g$. Then $t^0_{c_l} \leq t^0_g < t^1_{c_l}$, for any $1 \leq l \leq n$, holds since $t^0_g = t^0_{c_l}$ and the global clock has the smallest period so $t^0_g < t^1_{c_l}$. For the hypothesis, assume that given some $j$, cycle $i$ of clock domain $l$ is valid i.e., $t^i_{c_l} \leq t^j_g < t^{i+1}_{c_l}$. In the inductive step, when the global clock increments to $j+1$, if $c_l$ is not on a positive edge then $i$ is not incremented by the adder, and therefore the following inequality must hold:

$$t^i_{c_l} \leq t^{j+1}_g < t^{i+1}_{c_l}$$ (3.1)

This corresponds to the case where $c_l$ has not completed its cycle, thus $t^i_{c_l} \leq t^{j+1}_g$ holds since $t^i_{c_l} \leq t^j_g < t^{j+1}_g$. The inequality $t^{j+1}_g < t^{i+1}_{c_l}$ holds due to the following property of the global clock’s frequency. The period of any clock can be represented by an integer multiple of the global clock’s period. In the hypothesis, the smallest difference between $t^i_g$ and $t^{i+1}_{c_l}$ is one period of the global clock. Since a positive edge did not occur, the difference must have been greater than one period.

On the other hand, if $c_l$ is on a positive edge then $i$ is incremented by the adder. Thus, the following inequality must hold:

$$t^{i+1}_{c_l} \leq t^{j+1}_g < t^{i+2}_{c_l}$$ (3.2)

This corresponds to the case where $c_l$ has completed its cycle, and increments with the global clock. The inequality $t^{i+1}_{c_l} \leq t^{j+1}_g$ holds because $t^{i+1}_{c_l} = t^{j+1}_g$, and $t^{j+1}_g < t^{i+2}_{c_l}$ holds since $t^{i+2}_{c_l}$
is one period greater than $t_{ci}^{i+1}$ and therefore greater than $t_{gi}^{j+1}$.)

Since both inequalities hold it means that the synchronizer produces valid cycles.

Assume there exists a function $\lambda(x)$ that takes a bit vector as input and returns its integer representation (e.g. $\lambda((1,0,0)) = 4$). Let $s_{ci} = s_{ci}^{\lambda(x)}$ represent a multiplexer encoding where the output $s_{ci}$ is the $\lambda(x)$-th input of the multiplexer. This can be formally written as:

\[
\bigwedge_{j=0}^{m_{ci}-1} ( (\lambda(x) = j) \implies (s_{ci} = s_{ci}^{j} ) ) \equiv (s_{ci} = s_{ci}^{\lambda(x)} )
\] (3.3)

As such, the diagram in Fig. 3.1 can be represented by a 3QBF (i.e., a QBF with three quantifiers) BMC instance formally described as follows:

\[
\exists \text{cycle}_{ci}^S, \ldots, \text{cycle}_{ci}^S, s_{ci}^S, \ldots, s_{ci}^S, \text{edge}_{ci}^S, \ldots, \text{edge}_{ci}^S \forall gclk
\]

\[
\exists s_C, s'_C, \text{cycle}_C, \text{cycle}'_C, \text{edge}_C, \text{input}_C, \text{output}_C, \text{CDC}_C \mid
\]

\[
\bigwedge_{i=1}^{n} (I(s_{ci}^{0}) \land T_{ci}(s_{ci}, s'_{ci}, \text{input}_{ci}, \text{output}_{ci}, \text{CDC}_{ci})
\]

\[
\land (s_{ci} = s_{ci}^{\lambda(\text{cycle}_{ci})} ) \land (s'_{ci} = s_{ci}^{\lambda(\text{cycle}_{ci})+1} ) \land I(\text{cycle}_{ci}^{0})
\]

\[
\land (\text{cycle}_{ci} = \text{cycle}_{ci}^{\lambda(\text{gclk})} ) \land (\text{cycle}'_{ci} = \text{cycle}_{ci}^{\lambda(\text{gclk})+1} )
\] (3.4)

\[
\bigwedge_{j=0}^{m_g} (I(\text{edge}_{ci}^{j}) ) \land (\text{edge}_{ci} = \text{edge}_{ci}^{\lambda(\text{gclk})} )
\]

\[
\land (\text{edge}_{ci} \implies (\lambda(\text{cycle}'_{ci}) = \lambda(\text{cycle}_{ci}) + 1))
\]

\[
\land (-\text{edge}_{ci} \implies (\lambda(\text{cycle}'_{ci}) = \lambda(\text{cycle}_{ci})))
\]

\[
\land B(s_{ci}^{k})
\]

where $\text{CDC}_{ci} = \bigcup_{j=1}^{n} \text{CDC}_{ci,j}$, for all $0 \leq i \leq n$. In both existential scopes notation $s_C$ ($s'_C$) is used to denote variables $s_{ci} \ldots s_{ci} (s'_{ci} \ldots s'_{ci})$, and $\text{cycle}_C$ ($\text{cycle}'_C$) denotes variables $\text{cycle}_{ci} \ldots \text{cycle}_{ci}$ ($\text{cycle}'_{ci} \ldots \text{cycle}'_{ci}$). This definition similarly applies to $\text{edge}_C$, $\text{input}_C$, $\text{output}_C$, and $\text{CDC}_C$. Furthermore, $\text{gclk}$ is the global clock, $s_{ci}$ is the current state, $s'_{ci}$ is the next state, $\text{cycle}_{ci}$ is the current cycle, $\text{cycle}'_{ci}$ is the next cycle, $I(\text{cycle}_{ci}^{0})$, $I(\text{edge}_{ci}^{j})$, and $I(s_{ci}^{0})$ are the predicates recognizing valid initial constraints, and $\text{edge}_{ci} = \text{edge}_{ci}^{\lambda(\text{gclk})}$ represents a multiplexer encoding similar to Equation 3.3. Also, $s_C^{S} = \{ s_{ci}^{0}, \ldots, s_{ci}^{m_{ci}} \}$. Additionally, $\text{cycle}_{ci}^{S} = \{ \text{cycle}_{ci}^{0}, \ldots, \text{cycle}_{ci}^{m_{ci}} \}$, and a similar argument applies for $\text{edge}_{ci}^{S}$. Finally, $B(s_{ci}^{k})$ is a
bounded state at cycle $k$ for some $1 \leq l \leq n$, which is determined to be reachable or not in the BMC process.

When compared to the approach in [26], the proposed QBF-based formulation has several benefits. First, it is able to remove redundant time-frames by ensuring that the next state assignment for every cycle is only computed once. Furthermore, as experiments later in the chapter demonstrate, it is memory-efficient by orders of magnitude when compared to the one using an ILA approach. Due to the replacement of the ILA with universal quantification, designs with exceptionally long traces can be solved using the proposed formulation whereas it may not be possible to generate an ILA due to the excessive memory required [33].

3.3.1 Optimizing Performance

In some instances, a positive edge of the global clock will not imply a state transition for a clock because its positive edge has yet to occur. In this case $cycle'_{c_i} = cycle_{c_i}$ i.e., a state change doesn't occur. In this context, $T_{c_i}$ does not have to be computed because it will not result in a progression towards solving the problem.

Performance in a QBF framework can be optimized by adding a literal, $disable_{c_i}$, to every clause in $T_{c_i}$ which represents a path from a driver flip-flop to an input flip-flop clocked by the same clock. Then for every cycle of the global clock, an implication is added that sets the literal to True (i.e., turns off the transition relation), or sets the literal to false (i.e., allows the transition relation to be solved) depending on whether $edge^j_{c_i}$ is encoding a positive edge of $c_i$. This results in the following equation:

$$\bigwedge_{i=1}^{n} \bigwedge_{j=0}^{m_g} ((\lambda(gclk) = j) \implies (edge^j_{c_i} = \neg disable_{c_i}))$$

(3.5)

For clock domain crossing (CDC) paths, a literal is not added as it is assumed that multiple clock domain designs are implemented using asynchronous FIFOs or multi-flop synchronizers [43], or the clocks are already synchronized (i.e., the clocks do not drift and do not cause meta-stability issues.) These synchronizers contain minimal or no combinational logic, which would not substantially impact the solving time. If these paths were included, then the vari-
ables corresponding to them would have to be moved to the outer existential quantifier which has been shown to increase solving time \[28\].

### 3.3.2 Handling Latches

The formulation presented in the previous section can handle latches provided that inputs to latches are driven by flip-flops and/or clocks. This assumption is necessary because when a latch is enabled the data signal will behave as a clock \(i.e.,\) the latch will take the value of the data signal promptly. In the context of BMC, this would result in a clock signal (the data input to the latch) which can have an unbounded switching frequency. Between the period of one clock, the latch could have flipped values any number of times, which can theoretically require an infinite number of cycles to solve.

By applying the assumption, latched logic can be represented as a QBF-based ILA encoding that is clocked using a least common multiple frequency of the latch clock and the clock of the driver flip-flop. If the data input is treated as a clock, then a predefined input trace would be needed that specifies a periodic or non-periodic clock signal.

### 3.4 Divider-based Clock Synchronization

A potential overhead in the formulation presented in Section 3.3 is that when encoding a large number of cycles, the multiplexers may become large, which can potentially present a challenge to a QBF solver. Here, the proposed formulation is extended by replacing the multiplexer-based clock synchronizer with a divider-based synchronizer. The net benefit is a smaller and easier to solve CNF formula.

In the previous formulation, two outer multiplexers are used in the synchronization of the clock domains. Intuitively, this causes an overhead because, as the cycle bound for BMC increases, the number of paths in a multiplexer doubles with each new \(gclk\) bit. Furthermore, once a path in a multiplexer is chosen, all other paths become redundant. This may add unnecessary work as a QBF solver must assign the variables on the unused paths. This observation motivates for a modification using a divider-based synchronizer, as follows.
The basic idea of the divider-based synchronizer is to compute \( \text{cycle}_{c_i} \) by dividing the cycle of the global clock by some integer factor. In more detail, on every cycle of the global clock, one or more clock domains are changing states. Since all the clocks are periodic, a clock domain \( c_i \) changes states every time the global clock increments by a number of cycles, where this number is the factor mentioned above. For example in Fig. 3.3, every time the global clock increments by two cycles, \( c_2 \) increments by one cycle. Thus, if the current cycle of the global clock is divided by this factor (here, the factor is two) the result is the current cycle of \( c_2 \). Note that each clock domain may have a different factor which will be constant during a BMC run.

Due to this simplification, the divider formulation can only model designs which contain clocks that have a lowest common multiple and do not contain intentional phase shifting, such as cases where two or more clocks are the same frequency but contain 180 degree phase shifts. On the other hand, the formulation presented in Section 3.3 can handle these cases.

In the dashed boxes in Fig. 3.4, the hardware behind a QBF-based encoding for an ILA is
illustrated. The divider portion is presented to the left of these boxes. For each clock domain, the divider quotient is attached to the select lines of the multiplexers, and the global clock is set as the numerator. The factor, which is used as a denominator, is encoded as three constants which are the inputs illustrated to the left of each divider. The internal circuitry of the divider and how these three constants are computed is discussed in the next subsection.

By applying Theorem [3.3.2], the presented divider approach generates valid cycles. It also eliminates redundant time-frames because it only needs to compute the next state assignment for each clock cycle once.

Formally, the divider-based synchronizer in Fig. [3.4] can be formulated as a QBF instance by the following formula:

\[
\exists s^S_{c_1}, \ldots, s^S_{c_n}, mn_C, f_C \forall gclk \\
\exists s_C, s'_C, cycle_C, input_C, output_C, CDC_C | \\
\prod_{i=1}^{n} (I(s^0_{c_i}) \land T_{c_i}(s_{c_i}, s'_{c_i}, input_{c_i}, output_{c_i}, CDC_{c_i})) \\
\land (s_{c_i} = s^\lambda(cycle_{c_i})) \land (s'_{c_i} = s_{c_i}^\lambda(cycle_{c_i}) + 1) \\
\land T_{c_i}^{div}(gclk, 1, mn_{c_i}, f_{c_i}, cycle_{c_i}) \land I(mn_{c_i}) \land I(f_{c_i}) \\
\land B(s^k_{c_i}))
\]

where \(T^{div}_{c_i}(gclk, 1, mn_{c_i}, f_{c_i}, cycle_{c_i})\) is the combinational circuitry for the dividers in the dotted boxes, \(I(mn_{c_i})\) and \(I(f_{c_i})\) are the predicates recognizing valid initial constraints. Also, \(mn_C\) \((f_C)\) is used to denote variables \(mn_{c_1} \ldots mn_{c_n}\) \((f_{c_1} \ldots f_{c_n})\). All other notation remains as described in Section [3.3].

Fig. [3.5] shows a plot of memory vs. the number of cycles of the global clock. From this figure it can be seen that the size of the CNF of the formulation in Section [3.3] increases sharply, while the divider-based formulation increases at a much slower rate. The reason for this is for every new bit of \(gclk\) the size of the outer multiplexers in the original formulation doubles. This is in contrast to the divider-based formulation, where adding another input bit line requires attaching an additional one-bit divider to the original divider circuit. This addition does not cause the divider to double in size. Furthermore, if the number of cycles of the global clock
is not a power of 2, some paths will be left dangling. Satisfying these dangling paths requires extra computational effort. For example, if the number of cycles to be computed is 100, the select line will be 7 bits long giving a total of 128 cycles, 28 of which are redundant and will be left dangling. This CNF size increase may cause a slow down in a solver when tackling a design with large multiplexers.

3.4.1 An Efficient Divider Implementation

The following describes an efficient implementation of the divider circuitry shown in Fig. 3.4. General purpose dividers can be large when translated into CNF. As such, the divider is implemented using unsigned integer division with a constant number similar to the approach presented in [41]. By using unsigned integer division the divider can be implemented using a multiplier, an incrementer, and a shifter as illustrated in Fig. 3.6 which have simpler underlying circuitry. The idea behind this type of division is the fact that any number $n$ can be divided by $2^f$, where $f$ is some integer, by using a shifter. This allows division to be expressed as such:

$$\left\lfloor \frac{n}{d} \right\rfloor = \left\lfloor mn \ast \frac{n}{2^f} \right\rfloor$$

(3.7)
where \( d \) is the denominator, and \( mn = \frac{2^f}{d} \) is a magic number \([41]\). The magic number is a number which when multiplied by \( n \) then shifted right by \( f \) bits, produces the result of \( \frac{n}{d} \).

Integer operations are easier to compute than floating point operations. As a result, when computing the magic number, it is rounded down to an integer. This produces a dividing error which can be counteracted by incrementing \( n \) by 1. Finally, a value for \( f \) must be chosen, such that the error in the flooring operation in \( \lfloor \frac{2^f}{d} \rfloor \) is eliminated, and such that the magic number has the same width \( N \), as the width of the global clock. For this \( f = N + \lfloor \log_2 d \rfloor \) is chosen.

The magic number and \( f \) are precomputed as the denominator does not change. When the synchronizer is generated the implemented divider is:

\[
\text{cycle}_{c_i} = \lfloor (mn_{c_i} \times (gclk + 1)) \gg f_{c_i} \rfloor \quad (3.8)
\]

where the constant \( mn_{c_i} (f_{c_i}) \) is the \( mn \) (\( f \)) for \( c_i \), and \( gclk \) is universally quantified.

### 3.5 Experimental Results

This section presents experimental results for the proposed formulations. Experiments are run on a core i5-3570K workbench clocked at 3.40GHz, with a time limit of 7200 seconds. A memory limit of 4 GB is set. The backend QBF solving engine employed is RAREqS \([28]\), and the SAT solving engine compared to is the latest version of Minisat 2.2.0 \([23]\).

BMC is performed on six designs from OpenCores \([36]\), and one in-house real-life industrial
design. Each design has a varying number of clocks that ranges from 2 to 34. Both reachable and unreachable properties are tested. Table 3.1 displays the results for these seven designs. Under Column 1, the alphabetic prefix represents the name of the design, and the numerical suffix represents the test number. Columns 2-5 display the number of clocks, combinational logic gates, state elements, and latches, respectively. Finally, Column 6 denotes the number of cycles that are required to reach $B(s^k_{ci})$.

Four BMC formulations are profiled and compared to each other. The original clock unification method [26] is encoded in SAT and its results are shown in the columns titled uni-clk (SAT) (Columns 7 and 8). The proposed QBF-based clock synchronization and the extended divider-based formulations are outlined in the remaining columns titled multi-clk (QBF) and divider-based (QBF), respectively. In the final columns, the divider-based approach is augmented with the optimization from Section 3.3.1 and evaluated. For each experiment, the size of the CNF and the solving time are reported. The peak memory usage of each solver is analyzed later in this section. Finally, Columns 11, 14, and 17 display the memory savings for the proposed formulations when compared to the uni-clk (SAT) one.

From these numbers it can be seen that when compared to uni-clk (SAT), multi-clk (QBF) manages to have a memory savings average of 74% while keeping a comparable run-time, where uni-clk (SAT) runs for 32% of the run-time for multi-clk (QBF). The average slow down is computed individually, excluding the cases where the SAT-based approach runs out of memory (shown with $\infty$ in the table), and then averaging the results. Evidently, these memory savings are due to the elimination of the ILA and only using a single instance of the design (i.e., transition relationship) in the QBF-based approach. At low bounds for $k$ (range of 1...50, depending on the design), multi-clk (QBF) has the best memory savings, as the CNF of a 1...6 bit multiplexer is small. However, as the bound $k$ increases to greater than 6 bits the memory savings drop due to the generation of larger multiplexers. For these values of $k$, the run-time slows down due to the time required to assign the variables for unused paths in the outer multiplexers.

The divider-based formulation improves upon the original Section 3.3 formulation at higher values of $k$. The more moderate memory savings at low bounds is due to the multiplexers in
<table>
<thead>
<tr>
<th>Design Info</th>
<th>uni-clk (SAT)</th>
<th>multi-clk (QBF)</th>
<th>divider-based (QBF)</th>
<th>optimization (QBF)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Design</td>
<td>clks</td>
<td>Comb.</td>
<td>State</td>
</tr>
<tr>
<td>rsdecoder1</td>
<td>3</td>
<td>13543</td>
<td>526</td>
<td>5</td>
</tr>
<tr>
<td>rsdecoder2</td>
<td>3</td>
<td>13543</td>
<td>526</td>
<td>5</td>
</tr>
<tr>
<td>ethernet1</td>
<td>5</td>
<td>70139</td>
<td>10558</td>
<td>12</td>
</tr>
<tr>
<td>ethernet2</td>
<td>5</td>
<td>70139</td>
<td>10558</td>
<td>12</td>
</tr>
<tr>
<td>ac97_ctrl1</td>
<td>16</td>
<td>17591</td>
<td>2482</td>
<td>137</td>
</tr>
<tr>
<td>ac97_ctrl2</td>
<td>16</td>
<td>17591</td>
<td>2482</td>
<td>137</td>
</tr>
<tr>
<td>ac97_ctrl3</td>
<td>16</td>
<td>17591</td>
<td>2482</td>
<td>137</td>
</tr>
<tr>
<td>vga1</td>
<td>3</td>
<td>89402</td>
<td>17110</td>
<td>8</td>
</tr>
<tr>
<td>vga2</td>
<td>3</td>
<td>89402</td>
<td>17110</td>
<td>8</td>
</tr>
<tr>
<td>vga3</td>
<td>3</td>
<td>89402</td>
<td>17110</td>
<td>8</td>
</tr>
<tr>
<td>mem_ctrl1</td>
<td>10</td>
<td>48006</td>
<td>1239</td>
<td>94</td>
</tr>
<tr>
<td>mem_ctrl2</td>
<td>10</td>
<td>48006</td>
<td>1239</td>
<td>94</td>
</tr>
<tr>
<td>hpdmc1</td>
<td>34</td>
<td>9858</td>
<td>438</td>
<td>4</td>
</tr>
<tr>
<td>inhouse-uart1</td>
<td>2</td>
<td>3912</td>
<td>340</td>
<td>0</td>
</tr>
<tr>
<td>inhouse-uart2</td>
<td>2</td>
<td>3912</td>
<td>340</td>
<td>0</td>
</tr>
<tr>
<td>inhouse-uart3</td>
<td>2</td>
<td>3912</td>
<td>340</td>
<td>0</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>452</td>
<td>92.5</td>
<td>118</td>
<td>496</td>
</tr>
<tr>
<td>Impact</td>
<td>74%</td>
<td>32%</td>
<td>77%</td>
<td>35%</td>
</tr>
</tbody>
</table>
the original modeling being smaller than the dividers for low values of $k$. Both formulations perform similarly in run-time at low values of $k$. It can also be seen that multi-clk (QBF) saves about 50% more memory at low bounds when compared to [26]; however at higher bounds the divider-based formulation saves 20%-90%. It is worth noting that in BMC, the trace length $k$ is not known until the instance is solved. Thus the divider-based formulation is better due to its performance at higher bounds.

With the optimization presented in Section 3.3.1, the divider-based formulation becomes the most efficient of all proposed methods. There is a slight increase in the CNF size due to the added clauses, but a run-time improvement is observed due to eliminating the need to recompute the transition relations for cycles that have previously been evaluated. The performance benefits are less pronounced for circuits with similar frequencies because there are fewer redundant time-frames. Thus, transition relations are computed on a more frequent basis, and the overhead introduced by the added clauses exceeds the performance benefits. This can be seen with the inhouse-uart design, which has a clock that is half the frequency of the other.

In terms of run-time the SAT-based approach remains the fastest. Nevertheless, it is important to note that while the SAT-based approach takes 38% of the time that the divider-based formulation executes (if the mem-out cases are excluded), the divider-based formulation comes with a memory savings average of 76%, an attractive feature. Further, this QBF advantage is also demonstrated by entries in Table 3.1 which are infinity. It can be seen that in approximately 20% of the cases SAT runs out of memory when attempting to perform BMC, whereas the proposed QBF-based formulation does not. As work in QBF solvers remains a fertile research area, improvements in those engines will also benefit the proposed formulation.

Fig. 3.7 illustrates the memory size of the CNF (in logarithmic scale) for SAT-based BMC versus the QBF-based one with respect to the number of clock cycles in the problem instance for circuit ac97_ctrl. As it can be seen from the plot, the ILA explodes in size when the number of cycles exceeds 100. On the other hand, the QBF-based instances do not increase as rapidly because the cycles are encoded by the universal time quantifier. Fig. 3.7 shows that the original formulation fairs comparably with the divider-based formulation, saving more memory at lower
bounds due to the use of a small multiplexer. However, as the number of cycles increases, the size of the outer multiplexers also increases and its memory usage deteriorates.

Fig. 3.8 displays the solving time for the design vga, where the property was constrained at intervals of ten cycles. Results are again plotted in logarithmic scale while the graph is fitted to give the expected result at every cycle. The memory limit is relaxed to solve the instances. Due to the size doubling effect of the outer multiplexers when new clock cycle bits are added, the multi-clk formulation requires more computation time to process the multiplexers. The divider-based formulation does not grow as much because for every extra bit of the global clock, only a single bit adder and multiplier needs to be added. This results in a smaller CNF when compared to the multi-clk approach, producing a run-time performance gain that is comparable to the one by the SAT-based formulation, as the number of cycles increases.

Finally, Fig. 3.9 presents the peak memory usage when solving the problem. The designs are constrained for the first property in Table 3.1. Designs that are computed for a small number of cycles show little variation in memory usage, while designs with more cycles show a greater variation. This variation is primarily due to the increasing difference in CNF size between the QBF and SAT-based formulations as the bound $k$ increases.
Figure 3.8: Ratio of solving time to bound k for design vga (in log scale)

Figure 3.9: Peak memory usage when running each solver

3.6 Summary

This chapter introduces a novel QBF-based synchronization scheme for the verification of designs with multiple clock domains, the first of its kind. The original scheme is later extended
with the use of a divider-based synchronizer to further improve performance. Extensive experiments confirm the attractiveness of the approach as they demonstrate significant memory savings, with comparable run-times, when contrasted to the state-of-the-art.
Chapter 4

Revision Debug for Regression Verification

4.1 Introduction

To improve regression verification and the subsequent debugging process, the data returned by debugging must be preprocessed in a way that can provide confidence towards the cause of the error. Towards this direction, recent work in failure triage [38] has improved the process by implementing machine learning engines that determine which engineers are best suited to rectify a failure. Although useful, this work tries to cluster (i.e., bin) failures according to their impact in the functionality of the design, and it does not take into consideration other useful sources of information such as version control systems. These systems indicate what type of changes were implemented at a particular stage of the design process, after the last successful verification step. If this information is analyzed accurately, it can provide useful rankings for the candidate source of error(s) and aid the subsequent debugging process. Although failure triage shares similar objectives, its nature is complementary to the work presented here.

This chapter introduces a novel approach, which utilizes automated debug and existing information from version control systems, to rank design revisions using traditional machine learning techniques. The ranked revisions aid in determining which error location, originally identified by a debugger, is the actual error location that is responsible for a particular failure.
The prioritized revision(s), along with the failure(s), are sent to the design engineer(s) that committed the revision(s) to give them an intuition of what is causing the failure and when it was introduced. This step of analysis aims to ultimately reduce the number of debug iterations. The algorithm applies Support Vector Machine classification \cite{11} to first identify which revisions are bug fixes and which are incremental improvements. Then, a functional debugging step is performed and followed by Affinity Propagation \cite{25} clustering to rank potential error locations. A feature of this clustering algorithm is the ability to distinguish between failures caused by the same location and failures originating from different locations. Finally, a weighing scheme is utilized to identify the revisions which are most likely to be the source of the observed failure.

There are several additional benefits of using this algorithm in the verification flow. First, if a design previously passed verification but a new revision broke its functionality, the algorithm will identify the revision responsible for the failure. Next, as empirical results demonstrate, the intelligent parsing of the information as proposed here, triumphs over brute force script-based ranking approaches that are traditionally used in the industry. Specifically, when the brute force assigns a rank to the actual erroneous revisions, the ranking achieved by the algorithm presented here is on average 68% better than the brute force ranking. This arrives with an average run-time overhead of merely 4.631 seconds.

The remainder of this chapter is outlined as follows. Section 4.2 contains a presentation on related prior art, background material on functional automated debugging and a description of a brute force script-based approach, which draws comparisons with the algorithm presented in the chapter. Section 4.3 presents an illustrative discussion of the flow and intuition behind the algorithm. Section 4.4 gives a detailed breakdown of the revision debug algorithm. Finally, Section 4.5 presents the experiments and Section 4.6 concludes this work.

4.2 Preliminaries

4.2.1 Prior Art

There has been significant research in software verification that uses machine learning on version control systems to assist debugging. The authors of \cite{3} propose an algorithm for predicting
future code changes. Their algorithm analyzes recent revisions and predicts in which file a code change may occur. In \cite{32}, the authors propose an algorithm that can detect code patterns in version control systems. This allows them to identify potential bad coding styles. In the work of \cite{42}, the authors use Bayesian Belief Networks to predict how a code change will propagate to other modules in the software system. Finally, \cite{39} applies machine learning on version control systems to prioritize code review. The intuitive observation in that paper is that code which people have struggled with in the past is more likely to be bug ridden in the future.

There is far less work in regression debugging for hardware systems. To the best of our knowledge, the only related work is this of \cite{38} that implements a clustering-based failure triage engine to bin the errors after regression and help determine which engineers are best suited to rectify a failure. The authors introduce a data weighing scheme for simulation results, combined with modeling failures as multi-dimensional objects. Failure triage is performed by computing failure relations as distance metrics.

The above work applies machine learning algorithms to the results from traditional functional debug for regression verification. In contrast, in this work the information from version control systems is utilized to improve the overall process of coarse-grain debugging. Although our work presents a method to accelerate revision debug, it does not provide failure binning. As such, the work of \cite{38} is complementary to the one presented here.

### 4.2.2 Brute Force Approach

The following brute force approach is used to compare metrics of the algorithm presented later in the chapter. This approach can rectify the problem without using automated debug or machine learning, and is typical of what is happening today in the industry.

When regression returns a set of failures, the brute force approach is executed as presented in Algorithm 1. Each output error trace is compared to an expected golden trace, mismatching signals are passed to the algorithm. For every mismatch, the Breadth First Search algorithm is applied at the mismatching output signal to identify the fan-in cone as shown on Line 3. Each signal in this cone is then matched to their corresponding definitions in the HDL representation.
**Algorithm 1:** Brute Force Approach

**Data:** HDL Files $files$, Netlist $netlist$, Primary Outputs $PO$

**Result:** Potentially Erroneous Revisions $rev$

1. $hdlDef \leftarrow \emptyset$

2. **foreach** $po \in PO$ **do**

3. \hspace{1em} $nodes \leftarrow \text{BFS}(netlist, po)$

4. \hspace{1em} **foreach** $node \in nodes$ **do**

5. \hspace{2em} $hdlDef \leftarrow \{hdlDef \cup \text{mapNodeToDef}(files, netlist, node)\}$

6. \hspace{1em} **end**

7. **end**

8. $rev \leftarrow \text{vcsParse}(hdlDef)$

---

of the design as shown on Line 5. Finally, on Line 8 each revision in the version control system is parsed to see if they make any changes to these definitions and returned to the engineer if so.

Note that the matching HDL definitions will include the actual error location, as this is an over-approximation. However, it may include locations which cannot mask or fix the bug. Therefore, a percentage of the returned revisions may not be related to the failure.

### 4.3 Revision Debug Flow

This section contains an overview of our methodology and it is meant to give an intuitive understanding of the reasoning behind the algorithm presented in detail in the next section.

To improve coarse grain debugging, revisions are ranked based on their likelihood of causing a failure that was identified through regression. Revisions are added to a list in which high ranking revisions appear earlier in the list and low ranking revisions appear near the end of the list. This list, along with the failure(s), is presented to a verification engineer who must distribute the revision(s) and the failure(s) to the design engineers that created them for manual inspection. The design engineer uses the revision(s) to identify whether, when, and why they caused the failure(s). If the modifications applied by the revisions are determined to generate
erroneous behavior, then they are corrected. The overall ranking algorithm consists of three distinct phases.

In the first phase, presented in Fig. 4.1(a), the algorithm applies automated debugging \[45\] to find the potential error locations \((i.e., \text{lines of code in a RTL file})\) of the failures exposed during regression. Following this, the likelihood of each error location being an actual error is determined. For illustrative purposes, Fig. 4.2 explains this method using potential error location sets \(S_1 \ldots S_5\) for five failures \(f_1 \ldots f_5\) across two RTL code files. In this figure, each
axis represents a HDL file and the metric on each axis are line numbers in the file, *i.e.*, coordinate (1,1) is the first line in each file and the coordinates extend past the end of the files.

For each failure, the set of error locations which can correct the failure is mapped onto the graph, as seen in Fig. 4.2 as solid boxes to represent locations for possible correction. For example, assume that $S_1$ in Fig. 4.2 has two potential error locations in two files (represented by the $file_1$ and $file_2$ axis). One error spans lines 25-75 of the file along $file_1$ and the other spans lines 50-75 of the file along $file_2$. Hence, the area that represents $S_1$ in the graph would span coordinates from (25,50) to (75,75).

Intuitively, when two or more areas overlap, it indicates that the failures may be the result of the same error because debugging is indicating similar RTL code for correction. In Fig. 4.2 there are three different types of areas, which are denoted as A, B and C. Area A has no overlap, B is an overlap between two regions and C is an overlap between three. Area C represents RTL code that can be corrected to fix failures $f_1$, $f_2$ and $f_3$, while $f_4$ and $f_5$ can be corrected by modifying the right-most area B. For the sake of simplicity, if engineers were to correct the failures, they would target RTL code which, when corrected, eliminates most of the failures. Therefore, code which overlaps has higher priority for correction. In the figure, priority is given to code in overlap C, followed by B and then by A. In essence, the purpose of clustering is to
identify the regions that have the greatest overlap. In the figure, the dashed circles represent a cluster and the number of clusters indicate the minimum number of actual errors in the design. Revisions that make changes to code in the overlap are given a higher ranking.

The second phase of the algorithm, illustrated in Fig. 4.1(b), utilizes commit logs from version control systems to classify revisions as either bug fixes or not. One usually determines whether a revision is a bug fix by interpreting keywords from the commit log such as “fix,” or “corrected.” Thus, a classifier can classify revisions by utilizing unique words from commit logs as attributes for determining the classification. The intuition here is that a bug fix revision has a lower chance of being buggy when compared to revisions that make other changes. Thus, if a revision is a bug fix it would receive a lower ranking.

In the final phase, demonstrated by Fig. 4.1(c), revisions are ranked according to their likelihood of causing the observed failures using data gathered from the previous phases. Revisions that are not related to the failures are not included in the list.

The following section presents the details of each process in Fig. 4.1 respectively.

4.4 Generating the Ranked Revision List

4.4.1 Error Location Clustering

When regression is executed, each test exercises a portion of the design functionality. Failures may appear different because they can be detected through multiple mechanisms, such as assertions or a mismatch, as identified by a checker, between the observed and expected values. Despite this, they may have originated from the same location. The debugging results (potential error locations) for failures originating from the same error will be similar. This is in contrast to failures originating from different locations, which will have less comparable debugging results. The concept addressed here locates the similarities between failures so they can be used to aid in distinguishing between an actual error location and false positives.

Clusters represent the perceived number of actual errors in the design. Since the number of real errors is not known, an algorithm which does not require the number of clusters \( K \) to be known a priori, is needed. Affinity Propagation (AP) \cite{affinity_propagation} is a clustering algorithm which
can perform this task in the metric space. In traditional algorithms such as K-means, as demonstrated in Fig. 4.3(a), a priori knowledge of the number of clusters $K$ is needed because the objective is to minimize the distance between members and the $K$ cluster centers (denoted by $\mathbf{x}$ in the figure). In AP, clusters are created by maximizing the similarities between points and selecting an exemplar, a point which most closely represents the cluster. This is seen in Fig. 4.3(b) where members point towards the exemplar.

As discussed before, the potential error locations for each failure span an area in a space where dimensions are lines of an RTL file. However, the AP algorithm works with points in a space, therefore the data is preprocessed as illustrated by Fig. 4.4. In this figure, the solid boxes denote the potential error locations (lines of code in a set of files) for a particular failure ($S_i$) and dashed boxes indicate a subset of this area. The dashed arrows demonstrate how
a single location $s_{i_j}$ is converted to a point in space. The potential error location is a tuple $(\text{startLine}_{i_j}, \text{endLine}_{i_j}, \text{file}_{i_j})$, where $\text{startLine}_{i_j}$ (endLine$_{i_j}$) denotes the start (end) line of the possibly erroneous code in the file $\text{file}_{i_j}$. For each $s_{i_j}$ a vector $\vec{s}_{i_j}$ is created by assigning the axis spanned by $s_{i_j}$ to $(\text{startLine}_{i_j} + \text{endLine}_{i_j})/2$. Every other axis (where $\text{axisName}$ is the name of the file along that axis) is assigned to the following:

$$\text{average} \left( \frac{\text{startLine}_{i_j'} + \text{endLine}_{i_j'}}{2} \right), \quad \forall \, i', j' \mid \text{axisName} \in s_{i_j'} \land s_{i_j'} \in S_i$$

i.e., the average of each $s_{i_j'}$ in $S_i$ spanning the other axes. In essence, this ensures that vectors will be located around regions with overlap, with exemplars being those vectors located within the overlap.

**Example:** Suppose $s_{i_j}$ spans lines 25-75 on the $\text{file}_1$ axis and two other locations span lines 50-65 and 60-75 on the $\text{file}_2$ axis. To convert the area of $s_{i_j}$ to a point, the coordinate is given as $((75+25)/2, ((65+50)/2 + (60+75)/2)/2)$, i.e., $\vec{s}_{i_j}$ is (50, 62.5).

The AP algorithm determines clusters by maximizing the similarities between points in the cluster. The negative squared Euclidean distance similarity metric is applied, which allows the creation of clusters from those vectors that are in proximity to each other. Assuming $T$ is the total number of error locations returned by debugging, the similarity $sm(x_1, x_2)$ between two points is computed as follows:

$$sm(x_1, x_2) = -||x_1 - x_2||^2$$

(4.2)

where $x_1$ and $x_2$ are function parameters i.e., $x_1$ and $x_2$ can be replaced by any $\vec{s}_{i_j}$ to compute the similarity between the points. This function is used to create a $T \times T$ similarity matrix which calculates the negative squared Euclidean distance between $\vec{s}_{i_j}$ and every other point. Each vector has a preference associated with it that determines the probability of it being an exemplar. The preference for a vector is defined as the median of the set of similarities between that point and every other point.

Following the execution of the AP algorithm, the Euclidean distance between the cluster
exemplar and the potential error location is computed as follows:

\[ D(i, j) = ||exmp(label(s^j_i)) - s^j_i|| \]  

(4.3)

where \( exmp(x) \) is a function that takes a cluster label and returns the exemplar vector, and \( label(x) \) is a function that takes a vector and returns the associated cluster label determined by AP.

From the way the vector graph is constructed, the exemplar will be in an area with overlap. Since those areas closely represent the code which, when corrected, could fix multiple failures in the cluster. Hence, the closer another point is to the exemplar, the more likely it is to correct multiple failures. As such, the smaller \( D(i, j) \) is, the more it influences a revision to receive a higher ranking.

Example: The following illustrates how potential error locations are clustered and associated with revisions. Suppose automated debugging on one failure returns potential error locations on lines 25-75 and 50-75 in file\(_1\), and 25-26 in file\(_2\). Furthermore, suppose automated debugging on another failure returns potential error locations on lines 50-100 in file\(_1\) and 25-26 in file\(_2\). Thus, when this is graphed, an overlapping region exists between lines 50-75 in file\(_1\) and 25-26 in file\(_2\). Since there are five potential error locations across two failures, this gives us vectors \((50, 25.5), (62.5, 25.5), (56.25, 25.5), (75, 25.5)\) and \((75, 25.5)\) respectively. Suppose AP clustering returns one cluster containing all of these points, with the exemplar vector \((62.5, 25.5)\). This indicates that these two failures are closely related, and that lines 50-75 in file\(_1\) is the most likely error location that causes both failures. Thus, the actual code on lines 50-75 in file\(_1\) is matched to changes made by the revisions. Those revisions that are matched are given a higher ranking than revisions making changes to other code.

### 4.4.2 Revision Classification

To promote organization, engineers frequently employ version control systems that manage changes (i.e., revisions) to documents, computer programs and other pertinent information. Each revision usually targets information for a specific task. For example, a revision can be classified as adding functionality, modifying functionality, removing functionality, bug fixing,
etc. Since commit logs describe what changes an engineer has made in that revision, then it can be classified by applying its commit log to a machine learning classifier.

Revisions are classified through the use of Support Vector Machines (SVM). An SVM is an algorithm which predicts a classification of an object based on certain properties (features). For the methodology in this chapter, features are unique words in a commit log and revisions are classified as either a bug fix or not. Fig. 4.5 shows an SVM classification with two features. The dashed line is the classification boundary, indicating that Feature 2 is a word related to bug fixing (such as “bug”) and Feature 1 is a word unrelated to bug fixing (such as “implemented”). Each axis indicates the amount of occurrences of each feature within a commit log. Dots represent a classification of a revision with probabilities ranging from 100% bug fix to 0% bug fix (i.e., not likely bug fixes).

SVMs are chosen for two major reasons. First, SVMs can represent non-linear classifications through the use of kernel functions by mapping data into high dimensional feature spaces. Second, SVMs are capable of representing predicted classifications as probabilities. This aids in achieving a uniform ranking for cases where a revision includes both bug fixes and upgrades.

The classifier is taught to classify revisions through examples of revisions that are bug fixes and ones that are not. Once trained, it develops a prediction model that it uses to predict future classifications. For a set of $N$ commit logs, each is manually labeled as either bug fix or not, to construct a training label set $L = \{l_1, \ldots, l_N\}$. Following this, the number of $M$ unique words in the $N$ commit logs are used to construct a data matrix $C_{n,m}$ for $1 \leq n \leq N$ and $1 \leq m \leq M$, where each element is an integer counting the amount of times word $j$ appears in commit log.
i. In this sense, the labels and data matrix indicate examples of revision classifications. Once initial training has completed, any subsequent classifications only require the prediction model and the features of a revision commit log.

A critical inefficiency of SVMs is that they provide poor classification accuracy if $M >> N$ \[11\]. This is widely known as the “Curse of Dimensionality” where the number of dimensions exceed the number of samples. In the data matrix, it is undoubtedly true that $M >> N$, because commit logs can be of substantial variable length and word variety. Therefore, dimensionality reduction is applied to reduce the amount of unique words trained on, by removing words that appear too commonly or too infrequently in the commit logs.

Fig. 4.6 shows the number of occurrences of 4096 words in 2124 commit logs from eleven OpenCores \[36\] designs. From this graph, it can be seen that many words are used infrequently (such as specific email addresses) and only a hand-full that are common (such as “a,” “the,” “to,” etc.). Between the two dashed lines are the features that are retained for classification, while others are removed from consideration. This is determined by excluding features that match common words in the English language (if the commit logs are in another language, then common words of that language are used instead). Following this, the least occurring words that remain in the feature list are excluded iteratively until the final quantity of features respects the following constraint:

$$M \leq \frac{N}{2}$$ \hspace{1cm} (4.4)
Algorithm 2: Support Vector Machine Training

Data: Training Log Set \( trainLogs \), Testing Log Set \( testLogs \), Training Label Set \( L^{train} \), Testing Label Set \( L^{test} \)

Result: Prediction Model \( model \), Training Prediction \( pred_{tr} \), Training Accuracy \( acc_{tr} \), Testing Prediction \( pred_{te} \), Testing Accuracy \( acc_{te} \)

1. \( C^{train}_{n_1,m_1} \leftarrow \text{dimReduc}(\text{parseVCSLogs}(trainLogs)) \);
2. \( C^{test}_{n_2,m_2} \leftarrow \text{dimReduc}(\text{parseVCSLogs}(testLogs)) \);
3. \( model \leftarrow \text{svmTrain}(L^{train},C^{train}_{n_1,m_1}) \);
4. \{ \( acc_{tr}, pred_{tr} \) \} \leftarrow \text{svmPredict}(L^{train},C^{train}_{n_1,m_1},model) \);
5. \{ \( acc_{te}, pred_{te} \) \} \leftarrow \text{svmPredict}(L^{test},C^{test}_{n_2,m_2},model) \);

as this ratio gives reasonable prediction results \[15\]. The final number of features that are retained and used in the experimental section later on is 449, however this number can vary depending on the number of features and the language of the commit logs.

Algorithm 2 displays the SVM training algorithm that is executed one single time to obtain a prediction model used for subsequent classifications. A training commit log set is used to generate the prediction model and a testing commit log set is utilized to evaluate the prediction accuracy of the model. Initially, dimensionality reduction is applied to features obtained from the training and testing commit log set as shown on Lines 1-2. Next, on Line 3 the prediction model is generated. Finally, on Lines 4-5 the prediction accuracy of the model is evaluated.

Once the SVM learns a prediction model, the classification of subsequent revisions can be determined without knowing their classification. Their features are identified from the commit log and passed, along with the prediction model, to the SVM, which returns the prediction \( bugFix_i \), a real-valued probability number between zero and one. The algorithm which performs this is similar to Algorithm 2 except that it does not contain Lines 1-3 and Line 4.

Revisions which are bug fixes are expected to have a high probability, while those that are not, have low probability. Revisions which mix bug fixes and other activities are expected to fall in-between. It is anticipated that bug fixes do not introduce additional errors when compared
to significant code changes. As such, a higher bug fix probability will influence the revision to receive a lower ranking.

4.4.3 Weighted Revision Ranking

In the final phase, the data gathered from previous steps is applied to rank the revisions and associate each revision with a particular failure. The changes to RTL code made by a revision, denoted by \( R_l \) for \( 1 \leq l \leq N \) where \( N \) is the number of revisions, are gathered. Actual source code changes made by the revision are matched with the potential error locations (source code text) determined by debugging. If a match is found, then the revision is considered for ranking. The weight \( w_l \) of each revision is calculated and is used to determine the ranking of each revision. It is calculated as follows:

\[
w_l = \min_{i,j} \left( \frac{1}{2} \left( \frac{D(i,j)}{\max_{i,j}(D(i,j))} + \text{bugFix}_l \right) \right),
\]

\( \forall \ i, j \ | \ s_i^j \in R_l \) \hspace{1cm} (4.5)

This equation states that the lowest weight is given to a revision which is not a bug fix and matches a \( s_i^j \) closest to the exemplar in the cluster. Thus, the lower the weight of a revision, the higher the rank it receives.

Following this, \( K \) lists of ranked revisions are created, one for each cluster. Each list contains revisions that match the RTL code within the respective cluster. In this sense, each list contains the revisions that when modified can correct the failures in each cluster. The ordering of the lists is determined by sorting each revision by ascending weight.

Finally, the lists are merged together to create a unified list, a process illustrated by the following example. Suppose two lists \( A \) and \( B \) are generated, which are ranked revisions for two separate clusters. Another list \( C \) is the unified list created from \( A \) and \( B \). Also suppose each list is populated as follows:

\[
A = \begin{pmatrix} R_1 \\ R_2 \\ R_4 \\ \ldots \end{pmatrix}, \quad B = \begin{pmatrix} R_1 \\ R_3 \\ R_4 \\ \ldots \end{pmatrix}, \quad C = \begin{pmatrix} R_1 \\ R_2, R_3 \\ R_4 \\ \ldots \end{pmatrix} \hspace{1cm} (4.6)
\]
Observing list $C$, it can be seen that $R_1$ has the highest ranking and makes changes to code in both clusters i.e., it appears that it is responsible for all the errors. If an engineer determines that it is not the cause of the failures, then both $R_2$ and $R_3$ are equally considered. This happens because neither revision can be modified independently to correct all the failures. Thus, the ranking indicates that both revisions are likely erroneous and must be corrected.

The final unified list is presented to the verification engineer, who must manually inspect the revisions. It should be observed that due to the inclusion of all existing revisions and the completeness of automated debugging, at least one of the revisions in the unified list will be the cause of the failure(s).

4.5 Experimental Results

This section presents experimental results for the proposed revision debug framework. Experiments are conducted on a core i5-3570K workbench clocked at 3.40 GHz with 8 GB of RAM. Eight designs from OpenCores [36] and one in-house real-life industrial design are used to evaluate the methodology. Revision changes and commit logs are gathered from the OpenCores Subversion repositories, and are readily available with the design files. A total of eighteen tests are performed. The SAT-based automated debugger used to generate potential error locations for failures is implemented based on [45]. A platform is coded in Python that is used to parse debugging results and generate clusters with the AP algorithm. Classification predictions are performed without a priori knowledge of whether the revision is a bug fix or not. This is done by training a SVM prediction model prior to conducting the experiments. Furthermore, the sets of revisions used in experimentation and training are mutually exclusive. Finally, the results are used to rank the revisions for each test.

Errors are injected into the design by reading the commit log of the design and identifying actual bugs that the designer had to correct. A portion of these bugs are reintroduced into the design to test the methodology. For each regression run, a preexisting simulation test-bench is used to evaluate the functionality. Each regression run involves hundreds of input vectors. Erroneous responses are detected by comparing observed results to expected results from a
“golden” model checker. A separate script is coded to train the SVM and generate a prediction model. This is achieved by parsing revision commit logs and performing dimensionality reduction to overcome the “Curse of Dimensionality.” The features that are utilized for prediction model training are also used during the classification of experimental revisions.

Table 4.1 summarizes design information and statistics for each experimental run. Columns of this table show the design used for testing, an enumeration of the test runs, the number of standard cells (gates, DFFs, etc.) created by logic synthesis, the number of errors injected in the benchmark, the number of failures observed, the total number of error locations generated by automated debugging and the number of revisions stored by the version control system.

The effectiveness of the methodology is evaluated by comparing the ranking results of the methodology with the ranking obtained from the brute force approach described in Section 4.2.2. The comparison is performed by determining which approach gives a higher ranking to the revision(s) that are responsible for the actual errors in the designs.

When debugging returns a set of potential error locations, they are clustered. Fig. 4.7 shows the prediction accuracy of the AP clustering algorithm. The number of actual errors in the test are compared with the number of clusters. The prediction error is given as \( K - \# \text{actual} \).
## Table 4.1: Benchmark Statistics

| Design    | Test Num. | Logic | Num. Elem. | Logic Errors | Num. Fail. | $\sum_{i=1}^{\left|F\right|} |S_i|$ | Num. Rev. |
|-----------|-----------|-------|------------|--------------|------------|-----------------|-----------|
| Ethernet  | 1         | 6     | 76408      | 10           | 589        | 332             |           |
|           | 2         | 1     | 76408      | 4            | 880        | 332             |           |
| HA1588    | 3         | 4     | 9152       | 6            | 129        | 70              |           |
|           | 4         | 3     | 9152       | 6            | 69         | 70              |           |
|           | 5         | 7     | 9152       | 12           | 198        | 70              |           |
| I2C Core  | 6         | 3     | 3640       | 4            | 367        | 70              |           |
|           | 7         | 3     | 3640       | 12           | 178        | 70              |           |
| Tate Pairing | 8       | 4     | 106786     | 4            | 159        | 33              |           |
|           | 9         | 5     | 106786     | 37           | 81         | 33              |           |
| SD Card   | 10        | 1     | 38211      | 20           | 241        | 127             |           |
|           | 11        | 4     | 38211      | 36           | 179        | 127             |           |
| SDR CTRL  | 12        | 2     | 18374      | 5            | 1726       | 72              |           |
|           | 13        | 8     | 18374      | 10           | 1201       | 72              |           |
| 6507 CPU  | 14        | 2     | 9416       | 3            | 144        | 259             |           |
|           | 15        | 2     | 9416       | 3            | 75         | 259             |           |
| VGA       | 16        | 3     | 109797     | 5            | 173        | 59              |           |
| In-house  | 17        | 2     | 40197      | 16           | 42         | 177             |           |
| Packet Fwd.| 18   | 4     | 40197      | 23           | 51         | 177             |           |
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
Cluster Number
0 50 100 150 200
Number of Cluster Members

Figure 4.8: Number of members in each cluster for SDR CTRL (test 12)

errors for each test case. Ideally, the number of clusters will equal the number of actual errors and the prediction error will be 0. It can be seen that 9/18 tests have perfect cluster prediction, with an average of 4 cluster difference between the remaining tests. Observing test 12, it can be seen that 15 more clusters are created than needed. This is due to locating 1726 potential error locations across 9 different files, indicating that the two actual errors in this design were difficult to locate. Despite of this problem, the exemplars for each cluster provided adequate information as a 2/12 ranking was achieved for the revisions responsible for the errors. The reason for this is illustrated in Fig. 4.8, which shows the number of members in each of the 17 clusters of test 12. From this figure, it can be observed that clusters 8, 10, 11, and 15 are the primary influences for the ranking and potentially contain the actual errors in the design. The remaining clusters contain outliers with little overlap (i.e., false positive debugging solutions).

When training a prediction model, 2124 commit logs from eleven OpenCores design repositories, disjoint from the set used for experiments, are utilized as examples for classification training. From these logs, 4096 unique words are identified. After dimensionality reduction, this set is reduced to 449 commit log words that are used as features for classification. The 2124 commit logs are split into two sets, training and testing, where training receives 70% of
the logs and testing receives 30%. The training set is used to create the prediction model, and the testing set is utilized to test the effectiveness of that model. The entire training operation is a one-time process that takes 11.97 seconds, in which dimensionality reduction takes 7.81 seconds and prediction model training takes 4.16 seconds.

Fig. 4.9 shows the classification accuracy for training and testing revisions. The accuracy metric is computed as the percentage of the number of correct predictions in a class vs. the amount of samples in that class. Combined accuracy (i.e., number of correct predictions with classes combined vs. total amount of samples) was not taken due to the bias towards revisions not being bug fixes. Three different evaluations are performed for three cost settings. The cost is a variable in the SVM that determines the value of the weights in the model, effectively controlling the SVM learning rate and the prediction accuracy. A high cost will provide good training accuracy, but poor testing accuracy due to over-fitting, while a small cost will provide poor accuracy. From this figure, it can be seen that a cost of 75 is adequate as the testing accuracy is maximized for this value. There is a prediction accuracy mismatch between classes, due to bug fixes being characterized by a smaller word variety, when compared to other classifications that can be described with a much broader word variety.
Table 4.2 shows the results of revision ranking and presents a comparison of the ranking achieved by the proposed methodology vs. the industrial standard (brute force). From left to right, the columns of this table describe the test number, the rank and the time achieved by the brute force method, the rank with and without (denoted as w/o) the SVM classifier achieved by the proposed methodology. The next column displays the runtime of the proposed methodology with the classifier; however, the runtimes with and without the SVM classifier are comparable. In the final column, the improvement percentage is given, which indicates how much higher the ranking of the proposed methodology is when compared to the brute force method. This is calculated as $(1 - \frac{\text{Proposed Rank}}{\text{Brute Rank}}) \times 100\%$. From this table it can be seen that, on average, the proposed methodology achieves a 68.0% higher ranking than the brute force method. This arrives with a total amount of revisions that an engineer may need to look at, that is 34.0% of the total amount returned by the brute force method. This particular inefficiency of the brute force method is due to including revisions which are not related to the actual errors.

In Table 4.2 tests 14-16 give poor ranking results due to a combination of two factors. First, the erroneous revision contains bug fixes as well as an incremental upgrade and second, the upgrade itself introduces new errors. Since the erroneous revision made changes that could have been split up into separate clusters, then this results in a classification inaccuracy. Such inaccuracy gives the revision a 75% bug fix probability, negatively affecting the ranking. Despite of these setbacks, the ranking can be improved by only taking into consideration the results of automated debugging as clustering accuracy is not affected. In some tests, such as test 5, extra rankings are given because multiple revisions are erroneous. They make changes to code in similar locations, which means that their error locations appear in the same cluster. This results in each revision receiving a separate ranking.

Regarding run-time concerns, a majority of the time spent is in automated debugging, which takes between 400 to 3600 seconds. This process must always be performed during regression verification, regardless of whether it is manual or automated. In the case of manual debugging, the run-time is exceedingly longer than that of automated debugging. The overhead for performing clustering and ranking is between 0.6 to 30 seconds.
<table>
<thead>
<tr>
<th>Test Num.</th>
<th>Brute Force</th>
<th>Proposed</th>
<th>improv. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rank (rank/total)</td>
<td>Time (s)</td>
<td>Rank (rank/total)</td>
</tr>
<tr>
<td>1</td>
<td>58/129</td>
<td>0.235</td>
<td>6/27</td>
</tr>
<tr>
<td>2</td>
<td>84/129</td>
<td>0.244</td>
<td>5/10</td>
</tr>
<tr>
<td>3</td>
<td>11/12</td>
<td>0.231</td>
<td>1/5</td>
</tr>
<tr>
<td>4</td>
<td>11/26</td>
<td>0.447</td>
<td>1/7</td>
</tr>
<tr>
<td>5</td>
<td>15/32 &amp; 22/32</td>
<td>0.24</td>
<td>1/7 &amp; 2/7</td>
</tr>
<tr>
<td>6</td>
<td>23/23</td>
<td>0.783</td>
<td>1/13</td>
</tr>
<tr>
<td>7</td>
<td>15/23 &amp; 23/23</td>
<td>0.768</td>
<td>1/14 &amp; 3/14</td>
</tr>
<tr>
<td>8</td>
<td>16/19</td>
<td>0.087</td>
<td>4/6</td>
</tr>
<tr>
<td>9</td>
<td>13/19 &amp; 16/19</td>
<td>0.095</td>
<td>1/3 &amp; 2/3</td>
</tr>
<tr>
<td>10</td>
<td>19/32</td>
<td>1.482</td>
<td>4/9</td>
</tr>
<tr>
<td>11</td>
<td>19/32</td>
<td>1.445</td>
<td>4/10</td>
</tr>
<tr>
<td>12</td>
<td>19/26</td>
<td>1.484</td>
<td>2/12</td>
</tr>
<tr>
<td>13</td>
<td>19/26</td>
<td>1.381</td>
<td>2/13</td>
</tr>
<tr>
<td>14</td>
<td>6/64</td>
<td>0.183</td>
<td>35/49</td>
</tr>
<tr>
<td>15</td>
<td>27/64</td>
<td>0.173</td>
<td>41/48</td>
</tr>
<tr>
<td>16</td>
<td>3/18</td>
<td>0.305</td>
<td>11/15</td>
</tr>
<tr>
<td>17</td>
<td>17/83</td>
<td>0.166</td>
<td>2/15</td>
</tr>
<tr>
<td>18</td>
<td>17/83 &amp; 15/83</td>
<td>0.366</td>
<td>4/16 &amp; 8/16</td>
</tr>
<tr>
<td>AVG.</td>
<td>22/47</td>
<td>0.562</td>
<td>7/16</td>
</tr>
</tbody>
</table>

Table 4.2: Revision Ranking Performance
An interesting observation is that the size of the revision does not dictate whether the revision is more likely to be buggy, or more likely to be an upgrade. In general, including this metric when determining whether a revision is a source of an error, does not influence the ranking in any way. Intuitively, this can be explained because some designers debug their code extensively before committing, while others may not.

4.6 Summary

Regression verification remains to be a predominantly manual process. To ease verification, this chapter introduces a novel clustering-based revision debug framework. This methodology clusters automated debugging results and utilizes version control system data to rank revisions based on their likelihood of being responsible for the failures. Extensive experiments confirm the attractiveness of the approach as they demonstrate improved ranking capability, with negligible run-time overhead, when contrasted to the industrial standard.
Chapter 5

Syntax-Guided Re-synthesis for RTL Fault Correction

5.1 Introduction

In this chapter a syntax-guided synthesis tool for source-level error localization and correction is presented. It takes as input a buggy circuit design, at least one failing test vector, a set of correct test vectors, and a list of suspect error locations. Using this list and a library of rules characterizing typical source code mistakes, it automatically instruments the buggy design to allow each potential bug to either be left unchanged, or replaced with a set of possible corrections. The algorithm then combines the instrumented design with simulation vector(s) and solves a 2QBF-SAT (i.e., a QBF with two quantifiers) problem to find the minimum number of source-level changes from the original code which correct the bug. Because all correction rules describe semantically meaningful transformations, the changes presented to the user are highly likely to address the root cause of the error.

Experiments are conducted with 13 different benchmarks from 3 real-world designs available on OpenCores [36] and demonstrate that the methodology suggests valid corrections for 53% of the bugs within an average run-time of 70.93 seconds.
The remainder of this chapter is organized as follows. Related work is discussed in Section 5.2. Section 5.3 describes the algorithm in more detail. Experimental results are discussed in Section 5.4. Finally, Section 5.5 concludes the chapter.

5.2 Preliminaries

5.2.1 Prior Art

Debugging of hardware designs has been studied extensively in the previous three decades. This field typically focuses on two related but distinct facets of the problem: finding potential error locations, and proposing corrections that eliminate the failures.

*Error Localization:* Early works on design error localization were targeted at gate-level representations. Work by Chung et al. [18, 19] proposed a localization technique which expresses the problem as a set of Boolean equations, where the existence of a solution determines if the gate or wire is a potential error source or not. Smith et al. [45] improved on the scalability and quality of gate-level error localization using Boolean satisfiability (SAT).

Given the popularity of HDLs among hardware designers, source-level error localization has become increasingly attractive. Works byBloem et al. [13] and Peischl et al. [37] discussed Model-Based Diagnosis (MBD) methods for error localization. Several works [17, 45] adapted the concept of gate-level fault modeling to source-level error localization by mapping gates to their HDL description. The approach presented here uses the same concept of inserting multiplexers, but instead of having a single free signal, proper error corrections are inserted based on an error library model.

*Error Correction:* Error localization techniques usually generate a design component set: either RTL locations, gates in the netlist or combinational paths that can be modified to correct the error. Chang et al. [16] proposed an approach for correcting gate-level errors using signatures of candidate faulty gates. A signature is a list of bits each corresponding to the gate output for a given set of test vectors. Their approach corrects signatures and re-synthesizes them to replace the gate with one represented by the corrected signature. This work has been applied to source-level error correction and extended to hierarchical and sequential designs [17].
Jobstmann et al. \cite{29} suggested an approach to correct erroneous Verilog designs. It allows corrections that can be represented by arbitrary functions in terms of the state and input variables. This leads to a general correction model at the expense of readability and reasonability of correction suggestions.

5.3 Error Localization and Correction

Figure 5.1 shows the complete flow from a buggy RTL circuit to a list of suggested source-code correction(s) which fix the error(s) in the circuit. The buggy circuit must come with test vectors and at least one of them must be failing and expose the error(s). Evidently, the box labelled \textbf{Automated Debugging} represents error localization, while the remaining represents error correction.
The approach behind the algorithm is syntax-guided synthesis \cite{4}, where the original buggy RTL specification is tweaked in different ways. The purpose of this is to try to synthesize a new RTL specification which is syntactically similar to the buggy design yet which does not exhibit the error, and is therefore a candidate correction. In the spirit of syntax-guided synthesis, analyzing the syntactic template provided by the human designer, who inadvertently introduced the error in the first place, makes it possible to find good corrections more easily.

This section explains how the buggy circuit specification is instrumented, given a set of error rules in an empirical library.

### 5.3.1 Common Error Library

The key intuition of the approach is that most of the errors engineers make are predictable in nature. For example, one signal may be mistaken for another. Evidently, the algorithm presented here cannot capture all possible errors, but there is great practical value in efficiently capturing and correcting common errors and thus freeing precious designer time to concentrate on harder debug cases.

The common error library has been developed by reflecting on the experience of RTL designers and by manually inspecting a large number of buggy designs, including student assignments and bug fixes in open-source RTL repositories. At this point, the extensible library contains only general source-code transformation rules described in Table \ref{table:5.1}. Although limited, it turns out that this list is already effective.

### 5.3.2 Error Modeling

*Error rules* model common designer errors as modifications to the abstract syntax tree (AST) obtained by parsing the input RTL. Each rule is composed of two different parts: The first part, the rule checker, determines whether the particular rule is applicable. The second part, rule transformer, expresses the modifications to the AST necessary to include a set of potential corrections. For example, the rule checker of the rule in Figure \ref{fig:5.2} checks whether an AST node represents a bitwise OR operator. If the rule checker matches a particular node of the AST, the
### Table 5.1: A list of common error library rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Checker</th>
<th>Transformer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Signal indexing operation</td>
<td>Indices and ranges may be shifted to the left or right by one.</td>
</tr>
<tr>
<td>B</td>
<td>Incomplete case without default</td>
<td>All signals assigned in case get a default assignment of any compatible signal, or a pure free variable.</td>
</tr>
<tr>
<td>C</td>
<td>If ... If ... Else assigning the same signal</td>
<td>Allow use of a parallel If ... Else If ... Else with the same conditions.</td>
</tr>
<tr>
<td>D</td>
<td>Signal in any statement explicitly mentioned in candidate set</td>
<td>Allow referring instead to any compatible signal.</td>
</tr>
<tr>
<td>E</td>
<td>A bitwise comparison operator</td>
<td>Allow comparing with any other bitwise comparison operator instead.</td>
</tr>
<tr>
<td>F</td>
<td>A constant value on right-hand side; not an index/range</td>
<td>Allow using instead any constant value (a pure free variable).</td>
</tr>
<tr>
<td>G</td>
<td>A ternary expression</td>
<td>Allow using instead the same ternary expression, but with the condition inverted.</td>
</tr>
</tbody>
</table>
corresponding rule transformer is executed and the AST is modified to include a multiplexer that has the original operator and the rest of the operators as inputs, as shown in the figure.

Rule checkers can perform both structural and property checks. Structural checks are based on tree isomorphism (i.e., detecting if the structure of the AST subtree matches a reference one): they detect subtrees of interest and discard cases where the rule transformer should not be applied. Property checks are used to gather relevant non-structural information which is also needed to determine if the rule transformer should be applied, such as checking whether two identifiers in the matched subtree refer to the same constant value.

Rule transformers always instantiate the multiplexer structure illustrated in Figure 5.2, though not necessarily in the same AST location on which the rule matched. These multiplexers select an input depending on free variables. If an assignment to these free variables is necessary to correct the design, the 2QBF solver will find the required assignment.

As multiple rules may be triggered on the same AST node, it is proposed to apply the rules following a predetermined ordering roughly going from rules that are more specific to those that are more general. Although this case does not happen with the common error library described here, Table 5.1 is ordered by priority (the first rule is checked/applied first).
5.3.3 Instrumentation of the Buggy Circuit

To implement the error rules above, automated debugging is first performed to determine a set of possible error sources. Next, the frontend of Yosys [49], an open source framework for RTL synthesis, is modified to automatically instrument the buggy input circuit. A bottom-up, depth-first traversal of the AST is performed to trigger the code instrumentation. For each node in the traversal that matches an error source returned by debugging, each rule checker’s structural and property checks are performed around the AST location. They identify whether there is a rule in the common error library which can be applied. When a rule is triggered, the AST is modified to include the option of replacing or modifying the original AST with multiple potential corrections. Also, an extra primary output is added to the instrumented design, which is asserted when the number of non-zero free variables is less than a specified threshold. This allows one to additionally constrain the solver’s choice of free variables, not to only give correct behavior, but to employ the minimal number of changes necessary.

The next step is to construct a miter: a circuit where, through the solution of a particular satisfiability problem, one can determine a concrete value for all such free variables which render the circuit correct over all inputs.

5.3.4 Miter Construction

Although the basic construction of the miter is conventional for verification tasks, there are two aspects that differ here. First, it is assumed that a correct golden design is not available and that the error is exposed by an error trace used for functional simulation. Second, it is desired to control (and thus minimize) the number of individual corrections to the buggy code.

Figure 5.3 shows how the miter is built from the instrumented buggy circuit and a set of simulation traces, some of which expose the error. The test vector Content-Addressable Memory (CAM), shown to the right of the figure, is a look-up table that takes an input vector and returns the correct output response of the buggy circuit. If the CAM does not have an entry for a particular input vector, then it asserts the Miss signal. An extra multiplexer is added at
the output of the miter to ensure that the solver only tries to match the output for given input test vectors (as opposed to any input assignment). Thus, the miter is trivially satisfied (i.e., the primary output is 0) for all input stimuli not included in the subset of simulation traces that are considered. Accordingly, the miter is satisfied by a given vector of free variables (i.e., by a specific set of error rules correcting the error) when the functionality of the instrumented circuit matches the correct output response for all input stimuli in the restricted domain. One key advantage to using this construction as opposed to a golden reference, aside from the typically-limited availability of such a golden reference, is that it enhances scalability.

Besides the functional equivalence constraint, a second type of constraint is encoded to force a minimum number of corrections in the buggy RTL code. Figure 5.4 shows the logic responsible for this second check, annotated by the max rules check box. In this figure, the constant threshold value is set in successive runs of the 2QBF solver until the minimum number of changes is found. This allows the simplest source code modification(s) to be found, and rules out more obscure but legal solutions.
Once the formulation is constructed, it is converted to CNF, and a 2QBF (i.e., a QBF with two quantifiers) is devised. In this formula, \texttt{buggy} is the instrumented buggy circuit and \texttt{reference} is the set of correct input/output responses, and the formula is of the following form:

\[
\exists \vec{h} \forall \vec{x} : \texttt{buggy}(\vec{h}, \vec{x}) \Leftrightarrow \texttt{reference}(\vec{x})
\]  

(5.1)

where, \( \vec{x} \) is the input and the circuit \texttt{buggy} also takes a vector of free variables \( \vec{h} \). This is equivalent to answering the question, “does there exist a set of corrections (\( \vec{h} \)) that when enabled (assigned True), \texttt{buggy} and \texttt{reference} behave identically for all inputs \( \vec{x} \)?”

### 5.4 Experimental Results

The performance and scalability of the approach is evaluated on a range of Verilog benchmarks taken from OpenCores \[36\]. Each benchmark has one bug either injected artificially or taken from the version control history.

A commercial debugging tool \[45\] is used to obtain an initial set of error candidate locations in the input Verilog. This set is developed in the instrumentation process, as discussed in the previous section. At this point, the input Verilog is parsed and a netlist is produced by logic synthesis. Following this, the ILA is generated to handle sequential designs. This unrolled circuit is then passed to the CEGIS 2QBF SAT solver \[46\].

A miter is built from only three passing test vectors. The choice of three vectors is arbitrary,
and is a trade-off between avoiding trivial, incorrect solutions, and scalability. These vectors can be manually developed, provided by a software model, or be obtained from commercial design automation tools.

The results described below appear to validate the assumption that a few test vectors are enough to properly correct most circuits with this approach. Each correction found is exactly what a reasonable human designer would write, and fixes the bug most generally. In any case, the arbitrary use of three test vectors can be adjusted up if needed and does not compromise the generality of the approach.

Table 5.2 summarizes the experimental results. Columns 1 to 4, from left to right, display the buggy design used, the total number of free variables used in the instrumented design (including both the control signals of all multiplexers and all pure free variables representing constants), and the cumulative solver time spent on each experiment, the number of error candidate locations in the Verilog which need fixing (for the benchmarks where a correction was found). Columns 5 to 8 display whether the solver was actually able to find a solution with three or fewer changes, which rule(s) were essential to correct the bug, all rules which are employed in the instrumentation phase for each experiment, and whether the correction returned common corrections that any reasonable designer would do. Columns 9 to 13 illustrate the total number of AST nodes in the final circuit, what proportion of the AST was a descendant of an AST node in the error candidate set provided by the SAT-based debugger, how many AST nodes were matched by one of the rules, how many frames the original circuit was unrolled for in the pseudomiter, and finally, the relative size increase (measured as AND-Invert nodes by ABC) in the unrolled circuit vs. the golden reference design.

Note that even with the relatively sparse common error library, the algorithm was able to correct 53% of the simple bugs in the third-party designs with an average run-time of 70.93 seconds. Additionally, in the "Fixing Rule(s)" column it can be seen that one rule appears with striking regularity: rule D (see Table 5.1). This should come at no surprise, as this is one of the most general rules in the library.

Importantly, all of the solutions found were indeed common corrections that any reasonable designer would do. This supports the hypothesis that syntax-guided synthesis with common
<table>
<thead>
<tr>
<th>Buggy Design</th>
<th># Free Var. Bits</th>
<th>Total Solver Time (s)</th>
<th># RTL Changes</th>
<th>is Fixing</th>
<th>Applied Rule(s)</th>
<th>Sensible Sol.</th>
<th>Total AST Size</th>
<th>AST Marked %</th>
<th># Matched AST Nodes</th>
<th># Unroll Frames</th>
<th>Blowup</th>
</tr>
</thead>
<tbody>
<tr>
<td>spi_bug1</td>
<td>92</td>
<td>1.90</td>
<td>1</td>
<td>Yes</td>
<td>D</td>
<td>ABDEFG</td>
<td>2968</td>
<td>22.3%</td>
<td>20</td>
<td>20</td>
<td>2.94x</td>
</tr>
<tr>
<td>spi_bug2</td>
<td>8</td>
<td>1.69</td>
<td>–</td>
<td>No</td>
<td>BD</td>
<td>–</td>
<td>2964</td>
<td>4.9%</td>
<td>20</td>
<td>20</td>
<td>1.20x</td>
</tr>
<tr>
<td>spi_bug3</td>
<td>35</td>
<td>2.23</td>
<td>–</td>
<td>No</td>
<td>DEF</td>
<td>–</td>
<td>2968</td>
<td>13.3%</td>
<td>20</td>
<td>20</td>
<td>1.90x</td>
</tr>
<tr>
<td>spi_bug4</td>
<td>65</td>
<td>1.66</td>
<td>1</td>
<td>Yes</td>
<td>F</td>
<td>ABDF</td>
<td>2968</td>
<td>13.7%</td>
<td>13</td>
<td>20</td>
<td>2.14x</td>
</tr>
<tr>
<td>aes_bug1</td>
<td>373</td>
<td>18.71</td>
<td>–</td>
<td>No</td>
<td>ADFG</td>
<td>–</td>
<td>5080</td>
<td>5.2%</td>
<td>19</td>
<td>6</td>
<td>1.07x</td>
</tr>
<tr>
<td>aes_bug2</td>
<td>62</td>
<td>517.40</td>
<td>1</td>
<td>Yes</td>
<td>D</td>
<td>ABDG</td>
<td>5251</td>
<td>40.6%</td>
<td>33</td>
<td>6</td>
<td>1.29x</td>
</tr>
<tr>
<td>div_bug1</td>
<td>33</td>
<td>32.28</td>
<td>–</td>
<td>No</td>
<td>–</td>
<td>ADF</td>
<td>2486</td>
<td>14.1%</td>
<td>13</td>
<td>48</td>
<td>2.30x</td>
</tr>
<tr>
<td>div_bug2</td>
<td>20</td>
<td>71.47</td>
<td>2</td>
<td>Yes</td>
<td>DD</td>
<td>AD</td>
<td>2478</td>
<td>10.6%</td>
<td>8</td>
<td>48</td>
<td>2.12x</td>
</tr>
<tr>
<td>div_bug3</td>
<td>30</td>
<td>21.82</td>
<td>–</td>
<td>No</td>
<td>–</td>
<td>ADF</td>
<td>2486</td>
<td>13.6%</td>
<td>13</td>
<td>48</td>
<td>2.28x</td>
</tr>
<tr>
<td>div_bug4</td>
<td>26</td>
<td>78.90</td>
<td>1</td>
<td>Yes</td>
<td>D</td>
<td>ADF</td>
<td>2502</td>
<td>12.4%</td>
<td>10</td>
<td>48</td>
<td>2.24x</td>
</tr>
<tr>
<td>div_bug5</td>
<td>37</td>
<td>49.05</td>
<td>–</td>
<td>No</td>
<td>–</td>
<td>ADF</td>
<td>2516</td>
<td>14.7%</td>
<td>15</td>
<td>48</td>
<td>3.20x</td>
</tr>
<tr>
<td>div_bug6</td>
<td>32</td>
<td>17.75</td>
<td>–</td>
<td>No</td>
<td>–</td>
<td>ADF</td>
<td>2528</td>
<td>16.5%</td>
<td>20</td>
<td>48</td>
<td>1.99x</td>
</tr>
<tr>
<td>div_bug7</td>
<td>30</td>
<td>101.46</td>
<td>2</td>
<td>Yes</td>
<td>DD</td>
<td>ADF</td>
<td>2510</td>
<td>14.0%</td>
<td>12</td>
<td>48</td>
<td>3.15x</td>
</tr>
<tr>
<td>cpu_bug1</td>
<td>12</td>
<td>87.53</td>
<td>1</td>
<td>Yes</td>
<td>B</td>
<td>BDG</td>
<td>3842</td>
<td>20.1%</td>
<td>4</td>
<td>15</td>
<td>2.28x</td>
</tr>
<tr>
<td>cpu_bug2</td>
<td>46</td>
<td>60.05</td>
<td>1</td>
<td>Yes</td>
<td>C</td>
<td>CDEFG</td>
<td>3846</td>
<td>13.3%</td>
<td>5</td>
<td>15</td>
<td>2.56x</td>
</tr>
</tbody>
</table>

Table 5.2: Syntax-guided Error Correcting Experimental Results
error correction rules can be a particularly effective debugging tool. In practice, many but not all of the unsolved designs could be fixed by developing a posteriori additional rules, arguably general in nature. The extensibility of the common error library is a fundamental feature of the approach and the number of unsolved test cases is an artifact of the strict methodology.

Table 5.2 also includes extra information which can be useful in determining how practical the approach is and validating the use of the SAT-based debugger to compute an over-approximate error set. For example, in \texttt{div\_bug7}, only 14\% of the total AST nodes are considered, and yet the problem instance was over three times as large as the golden reference design. One can observe how the over-approximate error set is relatively large and thus of limited help to the designer (especially in contrast with the algorithm providing the simple and direct circuit correction). As rule \textbf{D} shows, the strength is in using fairly general rules, but this comes at a cost: without hints of where to look, one would be forced to use less general rules and fundamentally limit the ability to find bugs.

### 5.5 Summary

While extensive work has been done on functional verification and automatic debugging, more attention should be paid to the fact that humans are prone to repeatedly introduce the same bugs in their designs. In this chapter, the problem of error correction of a buggy RTL circuit was formulated as the problem of synthesizing a correct circuit with minimal syntactic distance from the buggy specification. Experiments demonstrate that the automated correction methodology corrects 53\% of the tests with an average run-time of 70.93 seconds.
Chapter 6

Conclusion and Future Work

6.1 Contributions

Verification has become a major challenge in the industrial design cycle. The time consumption of this process has largely been dictated by the amount of time debugging requires. Despite of the advancements in automated debugging and verification techniques, two major challenges remain that prevent a fully automated verification cycle. First, many problems in verification are computationally difficult, which poses problems as solving these problems may take an infeasible amount of time in the worst case. Second, the scalability of verification tasks remains to be a problem. This is due to the fact that a slight increase in design size may exponentially increase the amount of time required to solve the verification task.

This thesis presents three automated verification methodologies that attempt to improve the scalability and solving time of verification tasks. In the first contribution, a novel QBF-based multiple clock domain synchronization technique is presented. Second, a revision debug algorithm is presented that uses results from automated debug to direct an engineer towards revisions that may be responsible for the observed failures. In the final contribution, an automated correction methodology is presented that attempts to identify common errors that engineers make and to algorithmically correct these errors.

In more detail, the first contribution models sequential designs by replacing the notion of time through symbolic universal quantification. The design is then encoded using a QBF-based
ILA representation that keeps only one copy of the transition relation. The effect of this is to reduce the amount of memory usage vs. the amount used in traditional SAT-based encodings. Ultimately, this improves the scalability of verification tasks as it allows some problems to be solved, which are infeasible in SAT due to the memory requirements. Furthermore, as the QBF solver field grows, this methodology will benefit from the runtime improvements. Experiments utilizing this framework for BMC show memory reductions on average by 76% and in many cases, it is able to complete verification where SAT fails to do so.

The second contribution illustrates a novel algorithm that ranks design revisions that may be responsible for a particular failure following regression verification. This is accomplished using the results of automated debug and the data gathered from version control systems to link debugging solutions with revisions most likely to be a source of failure. The algorithm first applies clustering to identify potential error locations that are most likely to have caused the failure. Next, a Support Vector Machine classification algorithm is used to determine which revisions may have caused an error. Finally, a ranking scheme is used to prioritize revisions based on their probability of being a source of an error. Experiments demonstrate that the ranking achieved by the algorithm is on average 68% higher than the ranking achieved by an industrial script-based ranking approach, typically used in the industry today.

In the final contribution, a syntax-based automated correction methodology is detailed. This algorithm identifies typical source code mistakes from a library of common errors. It then instruments a 2QBF problem which determines whether a correction for the error can correct a failure, and returns the correction, if it determines that it corrects the failure. Experiments demonstrate that the automated correction methodology corrects 53% of the tests with an average run-time of 70.93 seconds.

6.2 Future Work

The contributions of this thesis rely heavily on the development of many formal techniques. Future research and development into these techniques will inevitably introduce ways to improve various verification tasks. Three areas for future research directions are given.
The first relates to the QBF-based multiple clock domain synchronization methodology. This methodology presents information that can be exploited by a QBF solver that is tailored for solving circuit CNFs. Towards that direction, research into a QBF solver that can detect different time-frames of a sequential design will improve solving time. This is because the variable assignments to a time-frame can be similar to the valid assignments for the next time-frame. As a result, if the QBF solver remembers previous assignments to time-frame variables, then it may use these assignments to make useful variable decisions in future time-frames.

On the other hand, the second direction relates to the revision debug methodology. This methodology will benefit from more information regarding which potential error locations is the actual error. In more detail, simulation metrics may aid in the quality of the returned revision list. For example, potential error locations, that are in close temporal proximity to the failure, may be more likely to have propagated the erroneous signal and caused the failure than locations far from the failure. Additionally, keeping track of which authors introduced errors in the past may aid in pinpointing future revisions which introduce errors.

The final research direction is towards improving the automated correction methodology. As a first step, increasing the amount of typical source code mistakes inside the library may increase the number of corrected errors. Towards that direction, a machine learning algorithm can be utilized to learn which types of errors an engineer is prone to create, utilizing data from version control systems. This will not only increase the number of typical source code mistakes in the library, but will also indicate which types of error are more likely to be made. The objective is in reducing the solving time of the methodology by attempting to fix the errors using the most likely corrections first.

As the fields of formal techniques and CAD tools develop and improve, they shall provide further exciting methodologies that improve verification tasks and reduce the time-to-market constraints. Ultimately, the consequence of this is the production of cheaper and reliable electronics in our ever developing digital civilization.
Bibliography


