Analysis of Single Event Upsets Propagation at Register Transfer Level in Combinational and Sequential Circuits Based on Satisfiability Modulo Theories

Ghaith Kazma

A Thesis
in
The Department
of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science (Electrical and Computer Engineering)
Concordia University
Montréal, Québec, Canada

May 2017

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By:	**Ghaith Kazma**
Entitled:	**Analysis of Single Event Upsets Propagation at Register Transfer Level in Combinational and Sequential Circuits Based on Satisfiability Modulo Theories**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Electrical and Computer Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________ Chair
Dr. R. Raut

______________________ External Examiner
Dr. J. Bentahar

______________________ Internal Examiner
Dr. S. Hashtrudi Zad

______________________ Co-Supervisor
Dr. O. Ait Mohamed

______________________ Co-Supervisor
Dr. Y. Savaria

Approved by: ______________________

Dr. W.E. Lynch, Chair
Department of Electrical and Computer Engineering

__________ 2017 ______________________

Dr. Amir Asif, Dean,
Faculty of Engineering and Computer Science

Abstract

Analysis of Single Event Upsets Propagation at Register Transfer Level in Combinational and Sequential Circuits Based on Satisfiability Modulo Theories

Ghaith Kazma

The progressive scaling of semiconductor technologies has led to significant performance improvements in digital designs. However, ultra-deep sub-micron technologies have increased the vulnerability of VLSI designs to soft errors. In order to allow a cost-effective reliability aware design process, it is critical to assess soft error reliability parameters in early design stages. This thesis proposes a new technique to model, analyze and estimate the propagation of Single Event Upsets (SEUs) in combinational and sequential designs described at the Register Transfer Level (RTL) using Satisfiability Modulo Theories (SMT). The propagation of SEUs through RTL bit-vector constructs is modeled as a Satisfiability problem using the SMT theory of bit-vectors.

At first, for combinational designs, two different analysis techniques, concrete and abstract modeling, are used in order to investigate the efficiency and accuracy of a data type reduction technique for soft error analysis. To analyze the vulnerability of the combinational circuits, we compute the Soft Error Rate (SER), which is a summation of the propagation probabilities. Concrete modeling uses two versions of the design, one faulty and one fault-free, in order to analyze SEU propagation. Abstract modeling uses a data type reduction technique to evaluate the difference in performance and accuracy over the first method. Experimental results demonstrate that the loss in accuracy due to abstract modeling depends on the design behavior. However, abstract modeling allows to reduce processing time significantly.

Following this first approach, the methodology is then extended to model and analyze SEU propagation in sequential circuits at RTL. In order to estimate the vulnerability of sequential circuits to soft errors, the methodology must be adapted to represent state transitions. To do so, we present an approach that uses circuit unrolling. This approach uses multiple unrolled copies of the design to represent the various state transitions. The fault propagation is then analyzed through a certain number of states. Useful information regarding the vulnerability to SEUs of the sequential circuit can then be generated. The propagation probabilities can be computed from the SEU injection cycle to multiple subsequent cycles. These results are then used to estimate the circuit Soft Error Rate (SER). Experimental results demonstrate the effectiveness and the applicability of the proposed approach.

Finally, we present a new methodology to estimate digital circuit vulnerability to soft errors at Register Transfer Level (RTL). Single Event Upsets (SEUs) propagation through RTL bit-vector operations is modeled and analyzed using a different modeling approach based on Satisfiability Modulo Theories (SMTs). The objective of this new approach is to improve the efficiency of the analysis. For instance, the bit-vector reduction operators and arithmetic operators were modeled in SMT to include the fault propagation properties. This approach uses only one copy of the design to do the analysis. This means that the fault propagation properties are embedded within the SMT equivalent of the RTL constructs themselves, and therefore does not require two-copies of the design to analyze. In order to illustrate the practical utilization of our work, we have analyzed different RTL combinational circuits. Experimental results demonstrate that the proposed framework is faster than other comparable contemporary techniques. Moreover, it provides more accurate and detailed results of the circuit vulnerability allowing a more efficient applicability of fault tolerance techniques.

To my parents

Acknowledgments

Firstly, I would like to thank Dr. Otmane Ait Mohamed for giving me the opportunity to pursue a Master's degree under his supervision. Dr. Ait Mohamed has taught me both professional and personal skills that will greatly benefit my career.

I would also like to thank Dr. Yvon Savaria for accepting to be my Co-Supervisor. His support was of great importance to the completion of my thesis. I am grateful to him for his encouragement and feedbacks.

Many thanks to all the Hardware Verification Group members for their support and encouragement. Every member, helped me in some way or another, throughout my Master's degree.

Finally, I would like to thank my family for their love and encouragement.

Contents

List of Figures **xi**

List of Tables **xii**

List of Acronyms **xiii**

1 Introduction **1**

1.1 Motivation 2

1.2 Thesis Contribution 5

1.3 Thesis Outline 6

2 Preliminaries and Related Works **8**

2.1 Single Event Upsets 8

2.2 State of the Art 11

2.2.1 Techniques at Gate Level 12

2.2.2 Techniques at Register Transfer Level 12

2.3 Boolean Satisfiability (SAT) 13

2.4 Satisfiability Modulo Theories (SMT) 14

2.5 Summary 15

3 Concrete and Abstract SMT Modeling and Analysis of SEUs Propagation in Combinational Circuits **16**

3.1 Introduction 16

3.2 Data Type Reduction 18

3.3 Proposed Methodology 20
3.3.1 Concrete Modeling 21
3.3.2 Abstract Modeling 24
3.3.3 Estimate SER due to SEUs 25
3.3.4 SMT Model Count 26
3.3.5 Examples 26
3.4 Experimental Results 32
3.4.1 Accuracy Analysis 32
3.4.2 Performance Analysis 34
3.5 Summary 36

4 SMT Modeling and Analysis of SEUs Propagation in Sequential Circuits 37
4.1 Introduction 37
4.2 Approximate Model Counting 38
4.3 SMT Modeling of Sequential Circuits 38
4.3.1 SMT Modeling of the Combinational Part 39
4.3.2 Unrolling of the Sequential Circuit 41
4.4 SEU Propagation and SER Estimation 43
4.4.1 Soft Error Rate Calculation 44
4.4.2 SEU Injection and Propagation 44
4.4.3 Soft Error Rate Computation 45
4.4.4 SMT Model Count 46
4.5 Example 48
4.6 Experimental Results 51
4.7 Summary 53

5 Efficient SMT Modeling and Analysis of SEUs Propagation in Combinational Circuits 55
5.1 Introduction 55

5.2 Modeling of RTL Constructs 56
5.2.1 Bit-Vector Signals 57
5.2.2 Logical Operators 57
5.2.3 Reduction Operators 61
5.2.4 Arithmetic Operators 64
5.2.5 If Statements and Case Statements 64
5.2.6 Concatenation, Extraction, and Extension 65
5.3 Analysis of SEUs Propagation and SER Estimation 65
5.4 Example 67
5.5 Experimental Results 70
5.5.1 Performance Analysis 71
5.5.2 Accuracy Analysis 73
5.6 Summary 76

6 Conclusion and Future Work 77
6.1 Conclusion 77
6.2 Future Work 78

Bibliography 79

Appendices 86

A Python SMT Library of RTL Constructs 87
A.1 Logical AND Operator 87
A.2 Logical OR Operator 88
A.3 Logical XOR Operator 88
A.4 Logical NOT Operator 88
A.5 Reduced AND Operator 89
A.6 Reduced OR Operator 89
A.7 Signed Extend Operator 89
A.8 Unsigned Extend Operator 90

A.9 Concatenation Operator 90

A.10 Extract Operator 90

A.11 Addition Operator 91

A.12 Subtraction Operator 91

A.13 Multiplication Operator 91

A.14 Division Operator 92

B Python Soft Error Analysis Scripts 93

List of Figures

3.1 Example of Fault Propagation in a 2-bit Multiplexer 20

3.2 Main Steps of the Proposed Methodology 21

3.3 Proposed Modeling of SEU Propagation Using the Fault-Free and Faulty Versions Model 23

3.4 RTL Structure of the 74L85 Benchmark That is a 4-bit Magnitude Comparator 27

3.5 RTL Structure of the 74283 Benchmark that is a Fast Adder Circuit . 30

4.1 Main Steps of the Proposed Methodology for the Analysis of SEU Propagation in Sequential Circuits 39

4.2 Methodology for Fault Propagation Analysis in Sequential Circuits Using Unrolling 42

4.3 Comparison of the Computation Time and SER Between Exhaustive Model Counting and SMTApproxMC 53

5.1 SMT Logical And Operator 58

5.2 Main Steps of the Proposed Modeling and Analysis of SEU Propagation at RTL 66

5.3 RTL Structure of the 74L85 Benchmark that is a 4-bit Magnitude Comparator 68

5.4 Comparison of the Computation Time Between the Boolean Based Modeling and the Proposed SMT Modeling for Different Sizes of Arithmetic Operations 73

5.5 Program Counter Datapath 75

List of Tables

1 SEUs Propagation Probabilities for the 74L85 4-bit Magnitude Comparator Circuit 29

2 Comparison Between the SER of the Concrete Modeling Technique and the Abstract Modeling Technique 33

3 Comparison Between the Computation Time for the Concrete Modeling Technique and the Abstract Modeling Technique 35

4 The Ratio of the Speed-Up in CPU Time over the Percentage Difference in Computed SER 36

5 SER of 4-bit Unsigned Up Counter with Synchronous Reset for Stage 0 and Stage 1 50

6 SER of 4-bit Unsigned Up Counter with Synchronous Reset for Stage 2 and Stage 3 51

7 Computation Times and SERs for ITC99 Benchmark Circuits 54

8 Logical Operators SMT Function Table 61

9 SEU Propagation Probabilities and Derived Vulnerabilities for the 74L85 4-bit Magnitude Comparator Circuit 69

10 Comparison of the Computation Time between the Proposed Methodology, the Technique Proposed in [1] and the Two-Versions Technique 72

11 Comparison of the Computed Soft Error Rate (SER) Between the Proposed Methodology, the Technique Proposed in [2] and the Two-Versions SMT Modeling 74

List of Acronyms

COTS	Commercial Off The Shelf
PRISM	Probabilistic Symbolic Model Checker
SAT	Boolean Satisfiability Problem
SEE	Single Event Effect
SET	Single Event Transient
SEU	Single Event Upset
SMT	Satisfiability Modulo Theories

Chapter 1

Introduction

A Single Event Upset (SEU) is a type of Single Event Effect (SEE). SEEs can be caused by galactic cosmic rays, cosmic solar particles or trapped protons in radiation belts [3]. SEEs induced by heavy ions, protons, and neutrons can seriously affect the reliability of electronic devices. For this reason, this has stimulated research on understanding and reducing the effects of SEEs by means of mitigation techniques [4]. Cosmic ray neutrons were recently found to cause errors even at ground level [5].

SEUs are a major source of soft errors in digital designs and modern electronic systems. Soft errors induced by SEUs have become one of the most challenging issues that impact the reliability of modern electronic systems. Digital circuits are more vulnerable to SEUs as the technology scales down. It is important to analyze and estimate the impact of SEUs on the behavior of digital designs.

In this section we present the motivation behind our work. We first discuss why there is a need to develop better and more efficient techniques to analyze the impact of SEUs in digital circuit early in the design cycle. We then discuss the contributions and the outline of this thesis.

1.1 Motivation

Soft errors induced by Single Event Effects (SEEs) such as Single Event Upsets (SEUs) and Single Event Transients (SETs) have become one of the most challenging issues that impact the reliability of modern electronic systems. The Soft Error Rate (SER) per chip has been reported to increase 100-fold from the 180nm to the 16nm CMOS technology nodes [6].

Aggressive technology scaling over the last several decades has made it harder for designers to guarantee the correct functionality of chips in the field. Although exponential growth in the number of transistors per chip has brought tremendous progress in the performance of semiconductor devices, it has increased their vulnerability to soft errors. Soft errors due to SEUs are one of the major reliability concerns in digital designs. There is, therefore, a growing need to analyze and estimate the impact of soft errors at an early stage in the design cycle.

A single particle hit on the state elements of a digital circuit can propagate through the circuit and affect the output at different clock cycles. Thus, in order to achieve cost efficient and reliable computing, it is crucial to take the reliability into consideration alongside the conventional area, power, and performance metrics in the design flow.

Moreover, analyzing soft errors in digital circuits is even more important due to the increased demand of commercial off the shelf (COTS) electronic components for avionics.

SEUs were responsible for the catastrophic failure and the recall of many safety critical systems. Such systems include implantable cardiac pacemakers [7] and implantable cardiac defibrillators [8].

This kind of failure has made it critical to analyze the vulnerability to soft errors and to apply efficient fault mitigation techniques to circuits used in critical systems. It is much more efficient for designers to apply fault mitigation techniques, such as Triple Modular Redundancy (TMR) [9], early in the design cycles.

Most contemporary techniques analyze the effect of SEUs and SETs at circuit level and gate level. SETs are affected by logical masking, electrical masking and latching window masking, while SEUs are only affected by logical masking. SETs are pulses with a certain duration and therefore, they can be masked due to electrical masking. For an SET or an SEU to propagate, it must be on a sensitized path from the location where it occurred to a primary output or a state element. This means that it should not be masked by the logical operations of the gates on its propagation path. For an SET, depending on the electrical properties of the gates it goes through on its propagation path, its duration (or pulse width) can be attenuated and masked before reaching a latch. If the pulse does not reach the state elements within the latching window, it is said to be masked by the latching window.

SEUs occur directly at the state elements, i.e., a bit-flip due to a direct particle hit, occurs at the latch itself. In this case, only logical masking can prevent an SEU from propagating to the primary outputs of a combinational circuit, or other state elements at subsequent clock cycles. Consequently, RTL descriptions of digital circuits provide sufficient information to analyze the propagation of SEUs, since only information regarding logical masking properties is required.

Early in the design cycle, digital circuits are defined at higher abstraction levels. This work deals with the soft error analysis of circuits described at Register Transfer Level (RTL). At RTL, circuit level details are not available and therefore this work focuses on the analysis of SEUs propagation in combinational and sequential circuits.

Conventionally, analyzing soft error glitches was done at circuit level, which requires a level of detail not available early in the design flow. However, applying mitigation techniques at later stages in the design cycle can be very costly. For this reason, there is an important need to develop techniques to analyze the impact of SEUs earlier.

At circuit level, parameters extraction and detailed simulations can provide a certain level of accuracy for phenomena such as electrical masking and SET width variation. However, when dealing with SEUs, this level of detail is not necessary. Moreover, the analysis at circuit level is very computationally intensive and would be intractable at the chip level. In other words, this type of analysis could be conducted on hundreds of transistors at most. Moreover, this type of approach are not applicable to RTL level constructs, due to a much higher level of abstraction. Therefore, the techniques at circuit level are not efficient to model SEU propagation at RTL.

At gate level, several techniques have been developed to analyze the effects of SEUs and SETs. Performing such analysis at the gate level requires the synthesis of RTL designs to gate level to be able to analyze their vulnerability to SETs and SEUs. Some of these techniques are very resource hungry and fail to analyze complex digital designs.

At RTL, the state of the art is lacking in techniques to analyze the soft error rates. At higher level of abstraction, the work focuses on analyzing the propagation of SEUs, since in RTL descriptions, loading and timing details are not available. Although some techniques allow the analysis of SEUs at high level, they still require the synthesis of RTL designs, due to their inability to model the higher level constructs used at RTL, e.g. *case* statements, *if* statements and linear arithmetic.

As we will see in Section 2.2.2, some techniques have recently been proposed to analyze SEU propagation at RTL. Researchers have proposed modeling approaches that use probabilistic model checking while other techniques use Boolean Satisfiability (SAT) modeling to analyze soft errors at RTL. Some of these techniques fail to handle moderately sized circuits while others cannot handle high level RTL constructs. Such techniques will still require the synthesis of the design from RTL to gate level in order to be applicable.

To summarize, due to the risks caused by SEUs in safety critical systems, it has become crucial to develop new techniques to analyze the impact of SEUs on digital circuits. This will give designers better insight regarding design vulnerability to soft errors early in the design stages. Most available techniques can only deal with the analysis of SEU propagation at gate level and circuit level. However, in order to apply efficient fault mitigation techniques, it is important to obtain Soft Error vulnerability in digital designs in early stages of design.

1.2 Thesis Contribution

In previous work, SEUs are analyzed at gate level and RTL through simulation based techniques and formal based techniques. Simulation based techniques often suffer from accuracy problems. In fact, even for small designs, simulation for all input vectors to study fault propagation becomes intractable which results in loss of accuracy.

In formal based techniques, several methodologies were proposed. Such techniques include modeling the SEU propagation at RTL as Discrete Time Markov Chains (DTMCs) such as [10] and [2]. Those properties are then verified against the constructed model using the PRISM model checker [11]. Some techniques use SAT Solvers to model the propagation of SEUs at RTL such as [1]. Using Pure Boolean SAT to model RTL constructs require to convert most RTL operations to Pure Boolean form. Therefore, those techniques require the synthesis of the design into a gate level netlist to be able to analyze the SEU propagation using a set of assertions.

In this work, we propose a new modeling and analysis approach to verify the propagation of SEUs using Satisfiability Modulo Theories (SMTs). Our approach is significantly faster than simulation based techniques while still offering great accuracy. Our technique is used to compute propagation probabilities and the Soft Error Rate (SER). The SER is essentially a measure to estimate the overall vulnerability of a circuit. Our work focuses on the logical masking property since at RTL, the information for electrical masking and latching window masking is not available. We model and analyze the fault propagation problem into a satisfiability problem using SMT and the Microsoft Z3 SMT solver [12].

We investigate the benefits and disadvantages of using a data type reduction technique with our proposed SMT modeling of SEU propagation. We also investigate the advantages of using SMT modeling over Pure Boolean SAT modeling when analyzing high level RTL circuits. SMTs allow to model RTL constructs into SMT format and verify the SEU propagation against a set of assertions. The advantage of SMT over

SAT, is that SAT based techniques require the use of synthesis tools, to take the RTL hardware description and produce a gate level netlist in order to apply the SAT level modeling.

We use our approach to analyze the vulnerability of combinational circuits at first, then the technique is extended and applied to sequential circuits. At first, the developed SMT model will use two copies of the design, one fault-free and one faulty version. The SEU is injected at one version and the outputs are compared to check fo fault propagation. For analyzing SEU propagation in sequential circuits, the combinational part of the design is unrolled to simulate the propagation of SEU through several clock cycles.

To improve our first modeling approach, we developed a library of SMT functions using Microsoft Z3's Python API [12]. The purpose of this new approach, is to create Python functions that use SMT operations to model the basic Verilog and VHDL RTL operations by including the fault propagation properties within the functions. This allows to directly model RTL designs without requiring two copies of the design.

In addition to the proposed modeling, we investigate the applicability of the proposed analysis by analyzing the vulnerability of several benchmark circuits from the ISCAS85 benchmarks [13] for combinational circuits and from the ITC99 benchmarks [14] for the sequential circuits.

To the best of our knowledge, techniques to model the propagation of SEUs at RTL using SMT have not been previously proposed. Our work was published in [15], [16] and [17].

1.3 Thesis Outline

The rest of this thesis is organized into four more chapters. In Chapter 2, we provide some preliminaries on the subject. We start by describing what SEEs are and discussing the difference between SETs and SEUs. Then we introduce the current state of the art at gate level and at RTL. Following this we describe what is Boolean

Satisfiability (SAT) and Satisfiability Modulo Theories (SMTs) and their applications.

In Chapter 3, we present our first approach at modeling combinational circuits using SMTs. To investigate the vulnerability to Soft Error, we use an approach involving two copies of the circuits under test. We also investigate the impact of a data type reduction technique on the accuracy and efficiency of the analysis. In order to do so, two modeling techniques are developed, concrete and abstract modeling.

In Chapter 4, we model and investigate the propagation of SEUs in sequential circuits. The modeling proposed in this chapter allows the analysis of SEUs propagation from each vulnerable node to the output of subsequent cycles of the sequential circuit. This is done by using multiple unrolled copies of the design to represent multiple state transitions of the sequential circuit. Moreover, the proposed modeling and analysis provides an early estimate of the Soft Error Rate (SER) of the design based on an approximate model counting approach.

In Chapter 5, we present a new modeling approach which embeds the fault propagation properties within the SMT model of the RTL design itself. In this approach, one copy of the design is used. For the analysis, an SMT model approximate counter is used to investigate the propagation of SEUs. Moreover, to investigate the efficiency of SMT versus Pure Boolean SAT in modeling RTL circuits, a comparison of the performance of the two modeling approaches is done on different sizes of arithmetic circuits.

Chapter 2

Preliminaries and Related Works

In this chapter we discuss the preliminaries and the related works. The goal of this chapter is to give the reader the required information to better understand the motivation behind our work and the tools required.

First, we define the Single Event Upset (SEU) and the logical masking effect. Next, we present the various state of art techniques at the gate level and Register Transfer level (RTL). We also provide an overview of Boolean Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) as well as their respective most popular solver engines.

2.1 Single Event Upsets

A Single Event Upset (SEU) is a type of Single Event Effect (SEE). SEEs can be divided into two main categories: Destructive SEEs and Non-Destructive SEEs, also known as Hard Errors and Soft Errors respectively. This work focuses on the Non-Destructive SEEs (Soft Errors) and more specifically on SEUs. The two main types of Non-Destructive SEEs are Single Event Upsets (SEUs) and Single Event Transients (SETs). Both of those single events change the state of a device without affecting its functionality. In other words, an SET or an SEU can potentially propagate to one or several outputs, thus causing an error, but the overall functionality of the circuit is not permanently affected.

In combinational logic, SETs are transient pulses generated in a gate that may propagate in a combinatorial circuit path and eventually be latched. SETs cause data on a wire to change logic for a short period of time. They are represented as a pulse with a certain duration and polarity, i.e., a pulse from logic `"0"` to logic `"1"` or logic `"1"` to logic `"0"`.

In memory devices, single event effects are called SEUs. An SEU is said to have occurred when a change in the state of a storage element such as memory or registers has occurred. SEUs have no pulse width, since they affect memory elements, when they occur, the value of the bit is flipped. On the other hand, in combinational logic, SETs are transient pulses that occur in a gate and may propagate through the combinational path and be latched by a flip-flop. SETs may potentially never be latched in a flip-flop if their pulse width is not strong enough.

The propagation of SETs through combinational designs is affected by logical, electrical and temporal masking (latching window masking) while the propagation of SEUs is only affected by logical masking. The low-level circuit details, that are required to analyze electrical masking and temporal masking, are not yet available in RTL descriptions of designs. Therefore, at RTL, the vulnerability of circuits can only be analyzed due to SEUs.

The effects that can prevent an SEE from propagating to an output are logical masking, electrical masking and latching window masking. It is important to note that all three effects can prevent an SET from propagating due to its nature, i.e., a pulse with a certain width. On the other hand, only logical masking affects SEU propagation, which are bit-flips that occur at flip-flops. The masking effects can be described as follows:

1. **Logical masking:** An SET or SEU can propagate if the input vector opens a sensitized path so that it reaches the design output.

2. **Electrical masking:** An SET requires a minimum width, which can be affected

by the gates on its propagation path, in order to reach the primary outputs or flip-flops.

3. **Latching window masking:** An SET can be latched by a register if it reaches the registers within the latching window with a large enough pulse width.

Soft errors occur when an SEU reaches the primary outputs of a design. SEUs can only be affected by logical masking. As mentioned earlier, logical masking can be described as follows: to cause an error, an SEU must propagate on a sensitized path from the location where it occurred to a primary output or a latch. When an SEU reaches the input of a gate, the fault will be logically masked if at least one of the other inputs of the same gate has a controlling value. This will prevent the SEU to reach any of the outputs and causing an error.

The same logic that is applied to these basic gate level operations is applicable to the equivalent logical operators at RTL. Different operators have different controlling logic and different logical masking probabilities. The *NOT* operator will always propagate an error since its output depends on only one input. The propagation probability for the *AND*, *OR*, *NAND* and *OR* operators depends on the number of inputs of the operator. The *AND* and *NAND* operators have a controlling logic of `"0"` while the *OR* and *NOR* operators have a controlling logic of `"1"`. For example, for an *AND* operator with 2 inputs, if one of the inputs has a logic value `"0"` (controlling logic), the output of the operator will be `"0"` regardless of the value of the other input. Therefore, an error at the other input will be logically masked. Furthermore, an SEU will always propagate through an *XOR* and *XNOR* operators.

Higher level synthesizable RTL constructs have an equivalent gate level representation. Therefore, any information regarding logical masking that is acquired from RTL descriptions of designs is equivalent to information gathered at the gate level. This is due to the fact that logical masking only depends on the logical operations of the circuit. The logical masking effect depends on the inputs, since for different input vectors, different paths are sensitized.

In the following chapters, the computation of the SEU propagation probabilities will be used to compute the Soft Error Rate (SER). We will explain in more details, how the propagation probabilities will be computed when analyzing some RTL benchmark circuits.

2.2 State of the Art

There has been considerable progress in functional verification of digital designs. However, the same cannot be said about non-functional verification. Non-functional verification investigates the behavior of designs in the presence of different uncertainties. Investigating non-functional properties is challenging due to the complexity of the modeling and the analysis. Moreover, many details about the uncertainties are not available at high abstraction levels.

In this section, we present the state of the art available at the gate level and RTL. Certain techniques analyze SEU propagation while other deal with SET propagation. It should be noted that, techniques that are used to model SET propagation can also be applied to analyze SEUs, since those techniques usually cover the effects of logical masking. The propagation of SET through combinational designs is affected by logical masking, electrical masking and temporal masking. As discussed in Section 2.1, logical masking occurs while the SET or SEU is propagating through a gate and at least one of the other inputs has a controlling logic value (e.g., `"0"` for a *NAND* gate). Electrical masking occurs when the duration of the SET pulse is less than the threshold of the gates on its path before reaching a latch. Temporal masking occurs if the SET pulse arrives at the flip-flop input outside of the latching window of registers.

On the other hand, SEUs occur at registers, and their propagation is only affected by logical masking, which is an effect that most technique that deal with SET propagation cover.

2.2.1 Techniques at Gate Level

Several techniques have been proposed at the gate level to deal with the analysis of SEUs. Those include simulation based techniques such as [18], [19], [20], [21] and formal based techniques such as [22], [23], [24]. More recently, formal techniques using an SMT based approach at gate level have been proposed such as [25].

Simulation based techniques have serious shortcomings as they are very time consuming for large circuits with many primary inputs. Furthermore, these techniques have their drawbacks in terms of accuracy. This is mainly because their accuracy is determined by the ratio of the simulated sample size over the total vector space size. These approaches have a scalability problem and cannot be applied on all types of designs.

The problem with the techniques developed to analyze SEU and SET propagation at gate level is that they cannot be applied at earlier stages in the design. RTL descriptions of digital circuits contain higher level constructs such as *if* statements, *case* statements and linear arithmetic operators that cannot be handled by the above techniques. In the next section, we present a literature review on the current techniques that deal with SEU propagation at RTL.

2.2.2 Techniques at Register Transfer Level

At Register Transfer Level (RTL), there is far less techniques to analyze SEU propagation. However, some techniques to analyze SEU propagation have been proposed. Those include fault simulation techniques such as [18] [26] and formal verification methods such as [10], [1], [2].

However, the techniques to analyze SEU propagation at gate level are not applicable at an early design stage. Moreover, simulation techniques at RTL are very time consuming. Even formal based techniques fail to handle moderately sized circuits. For instance, existing formal based techniques, such as Reduced Ordered Binary Decision

Diagrams (ROBDDs) [23] and Multi-Terminal BDDs (MTBDDs) [2], are computationally expensive i.e., suffer from a state explosion problem.

Formal verification based techniques are resource hungry when used to analyze a complex digital design at RTL. For example, formal techniques run out of memory, even when trying to analyze moderate size designs, e.g., a 14-bit adder [10].

Some techniques have been developed to analyze SEU propagation at RTL using Pure Boolean SAT such as [1]. Those techniques require the conversion of the SEU propagation in a behavioral description into an instance of a SAT problem. In order to do this, they require intermediate steps, since pure Boolean SAT cannot support various RTL constructs. The RTL description of the circuit is modified to include the fault propagation properties using a method that uses two copies of the design and compares the outputs using an *XOR* operation. The resulting RTL description is then converted to an equivalent conjunctive normal form (CNF) which is sent to a SAT solver. The CNF is the equivalent SAT Boolean function converted to a product-of-sums. It contains the logical *AND* of the set of clauses that represent the formula. This conversion requires two intermediate steps. The modified RTL Verilog description is converted to SMV format using the Cadence SMV tool [27]. Then tools in the AIGER library [28] were used to convert from SMV format to CNF. The resulting CNF file can then be sent to a SAT solver.

In this work we overcome this limitation by using SMT to model SEU propagation. The basic RTL constructs can be modeled directly into SMT format which does not require intermediate conversions steps.

2.3 Boolean Satisfiability (SAT)

SAT is an abbreviation for the Boolean Satisfiability Problem. The SAT problem is the problem of determining whether theres exists a variables assignment such that a given propositional formula evaluates to True. In other words, the problem is to determine whether values (True or False) can be assigned to the variables of the formula for it to be True (satisfiable). If no such assignment can be found, then the formula is said to be UNSAT, or unsatisfiable.

The problem of Boolean Satisfiability (SAT) is not only of interest in computer science, since it has also received great attention in other areas where it has seen significant practical applications [29].

Boolean Satisfiability (SAT) is often used an increasing number of applications in Electronic Design Automation (EDA) as well as many other engineering fields [30]. More specifically, SAT has been used to formulate EDA problems such as test pattern generation, circuit delay computation, logic optimization, combinational equivalence checking, bounded model-checking and functional test vector generation.

Some well known SAT solvers include: MiniSat [31], MiniSat2 [32], PicoSAT [33], RSat 2.0 [34] and Glucose [35].

2.4 Satisfiability Modulo Theories (SMT)

The advent of Satisfiability Modulo Theories (SMTs) [36] solved the problem of being restricted to a pure Boolean representation, which is not efficient and sometimes inadequate when representing several classes of systems. SMT is an extension of the SAT decision problem, where the formulas are expressed in first-order logic, with associated background theories. SMTs have been used to model and solve software and hardware engineering problems. In this work, we show how linear arithmetic and bit-vector theories can be used to model SEUs propagation.

SMT solvers that support the theory of bit-vectors provides concrete models for bit-vector operations. Bit-vectors can be used to model designs directly on the word-level. This proves to be useful when converting bit-vector RTL operations to SMT operations.

High level behavioral and structural RTL descriptions of circuits include a variety of RTL operations not available at gate level. This includes arithmetic operations

operations, *if* statements and *case* statements. SMT solvers support bit-vector arithmetic operations such as addition, subtraction, multiplication and division. Moreover, they support *If-Then-Else* statements (*ITE*).

Some modern SMT solvers include: Microsoft Z3 [12], Boolector [37], Yices [38], Yices2 [39] and CVC4 [40].

2.5 Summary

In this chapter, we discussed the preliminaries required to better understand our proposed methodology. We described SEUs and their effects on modern digital circuits. Then we discussed the related works dealing with the analysis of SEUs both at the gate level and RTL level. There are far more techniques developed to tackle this problem at gate level. We described Boolean Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) in order to understand the main differences and our motivation behind using SMT to model the problem of SEU propagation.

Chapter 3

Concrete and Abstract SMT Modeling and Analysis of SEUs Propagation in Combinational Circuits

3.1 Introduction

In this chapter, a methodology is proposed to analyze Single Event Upset (SEU) propagation in combinational designs described at Register Transfer level (RTL). In this approach we use a well-known technique that utilizes two copies of a given design, one fault-free and one faulty, where an SEU is injected. The accuracy of the two-versions modeling approach was proven in [1]. The outputs of both copies are then compared to check if they are different. In that case, the SEU propagated to the output and caused an error.

Our methodology utilizes Satisfiability Modulo Theories (SMT) to model SEU propagation in RTL designs as a Satisfiability problem. We propose two modeling approaches to analyze the vulnerability of RTL designs to SEUs, i.e., concrete modeling and abstract modeling.

1. **Concrete Modeling of SEU Propagation:** In this approach, the generated SMT models preserve the full functionality of the RTL description and is modeled using the SMT theory of bit-vectors. Two copies of the generated SMT model are used, the first is a fault-free version and the second is a faulty version where an SEU is injected. The outputs of both circuits are compared to check if the SEU caused an error at the output.

2. **Abstract Modeling of SEU Propagation:** In this approach, based on a SEU injection scenario, the SMT model is reduced to improve the scalability by adapting the data type reduction technique used in [2]. This technique is a form of abstraction which comprises two main elements. First, bit-vectors are reduced to 1-bit signals and second, operators are abstracted with a simplified error propagation model. As shown later, this simplified error propagation model is often quite pessimistic because it predicts propagation over whole bit-vectors rather than specific bits.

The proposed technique starts by translating the Verilog RTL behavioral description of the design into an SMT equivalent model. RTL operations such as logical, reduction, arithmetic, *if* statements and *case* statements are converted into their SMT equivalent using the theories of bit-vectors and linear arithmetic. Then, based on the adopted modeling approach, the proposed methodology allows the analysis of SEUs propagation using a set of assertions. To estimate the vulnerability of digital circuits, we compute the Soft Error Rate (SER). The SER is a summation of the propagation probabilities of SEUs injected at individual bits, therefore it can be greater than 1.

Experimental results demonstrate that the loss in accuracy due to abstract modeling depends on the design behavior. For example, for some circuits, the loss in accuracy (percentage difference between the computed SERs) was around 187%, while for other circuits it was as low as 0.03%. The percentage of error for individual injection scenarios cannot exceed 100% since the propagation probabilities range from

0 to 1. Since we are computing the percentage difference between the SERs, our values can exceed 100%. However, abstract modeling allows reducing processing time significantly and an average reduction factor of 67.33 is reported. The reported results demonstrate that the abstract modeling technique yields considerable speed-up in computation time at a certain cost in accuracy. The generated results are used to investigate the trade-off between the speed-up of abstract modeling and the accuracy of concrete modeling.

The rest of this chapter is organized as follows. Section 3.2 explains the data type reduction technique used and how it affects the accuracy of SEU propagation analysis. Section 3.3 explains our proposed modeling and the analysis of SEUs propagation at RTL. In Section 3.4, we explain our experiments and results. Section 3.5 concludes by summarizing the analysis and experiments.

3.2 Data Type Reduction

Existing formal analysis methods can be combined with different reduction techniques to improve their scalability. SEUs propagation behavior at RTL can be reduced using a data type reduction technique. The data type reduction technique proposed in [2] is adapted to our SMT model to investigate its effect on our modeling approach. In this work, we define the data type reduction technique used as follows:

1. For a bit-vector signal, if one bit is faulty, then the whole vector is considered faulty. In other words, a multi-bit signal is considered as a 1-bit signal.

2. SEU propagation through register transfer operations such as addition and multiplication is modeled to transparently propagate from inputs to outputs. This modeling loses track of the exact location of faults. For example, in an addition involving two input n-bit signals **A** and **B** and output **C**, a fault occurring at bit k of signal **A**, cannot propagate to bit i of output **C**, given $i < k$. These details would be abstracted when using a data type reduction technique. A

faulty input signal in an addition operation will automatically imply a faulty output, without providing details regarding specific bit vulnerabilities.

3. For a *case* statement or a block of *if* statements, which represents a multiplexer, if the selection signal is faulty, then the output is considered to be faulty as well, regardless of the values of the input signals. If one of the input signal is faulty, the output will be faulty if the selection signals currently select the faulty input.

The main issue with the utilization of such a reduction technique is that we lose track of the exact error location. In other words, it is not possible to identify the specific faulty bits in the bit-vector signals.

To better understand the impact of such modeling, consider the RTL signal assignment shown in Listing 3.1. The bit-vectors **next_addr** and **start_addr** are 32-bit signals.

```
next_addr = start_addr + 4;
addr_msb = next_addr[31:16];
addr_lsb = next_addr[15:0];
```

Listing 3.1: RTL Assignment Example

Instead of performing an SEU injection at every bit of the 32-bit **start_addr** signal, the signal is reduced to a single bit. It is assumed that if **start_addr** is faulty, then **next_addr** is faulty.

However, a certain output of the circuit may only depend on the value of certain bits of the **next_addr** signal while the fault is present in some other bits. In Listing 3.1, the **addr_msb** signal depends on the uppermost 16 bits of **next_addr** while **addr_lsb** depends on the lower 16 bits. In this case, the output may mistakenly be considered faulty for several SEU injection scenarios.

As another example, consider the 2-bit RTL multiplexer shown in Figure 3.1.

If the select signal **sel** is faulty, then the output **y** is considered as faulty. However, in this example, a faulty **sel** signal can cause at most 2 bits of **y** to be faulty.

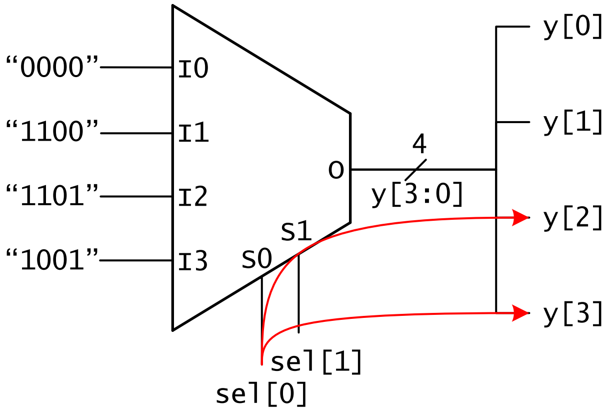


Figure 3.1: Example of Fault Propagation in a 2-bit Multiplexer

If **sel[0]** signal is faulty, then this fault will only propagate to **y[2]** and **y[3]**. The propagation of the fault is conditional on having an active path from the **sel** signal to the output i.e., the fault is not logically masked. Given **sel[0]** is faulty, an error will occur at **y[3]** only if **sel[1]** is `"0"`. On the other hand, a fault at **sel[0]** will always propagate to **y[2]**, regardless of the state of **sel[1]**. Similarly, a fault at **sel[0]** will never propagate to outputs **y[0]** and **y[1]**. However, this level of detail is fully abstracted when using this data type reduction technique. Therefore, the computed vulnerability of the circuit is inaccurate.

3.3 Proposed Methodology

The proposed methodology introduces a new formal modeling and analysis of SEU propagation at RTL using Satisfiability Modulo Theories. In this section, we propose two modeling approaches: concrete and abstract modeling.

The main steps of both approaches are presented in Figure 3.2. For both modeling techniques, the RTL signals and operations are converted into SMT format. The SMT solver used to model and analyze the propagation of SEUs in the RTL circuits given is Microsoft's Z3 [12]. In order to automate the analysis steps such as fault injection and

SEU propagation probability computation, we used Python scripting. Microsoft's Z3 provides a Python API for such application.

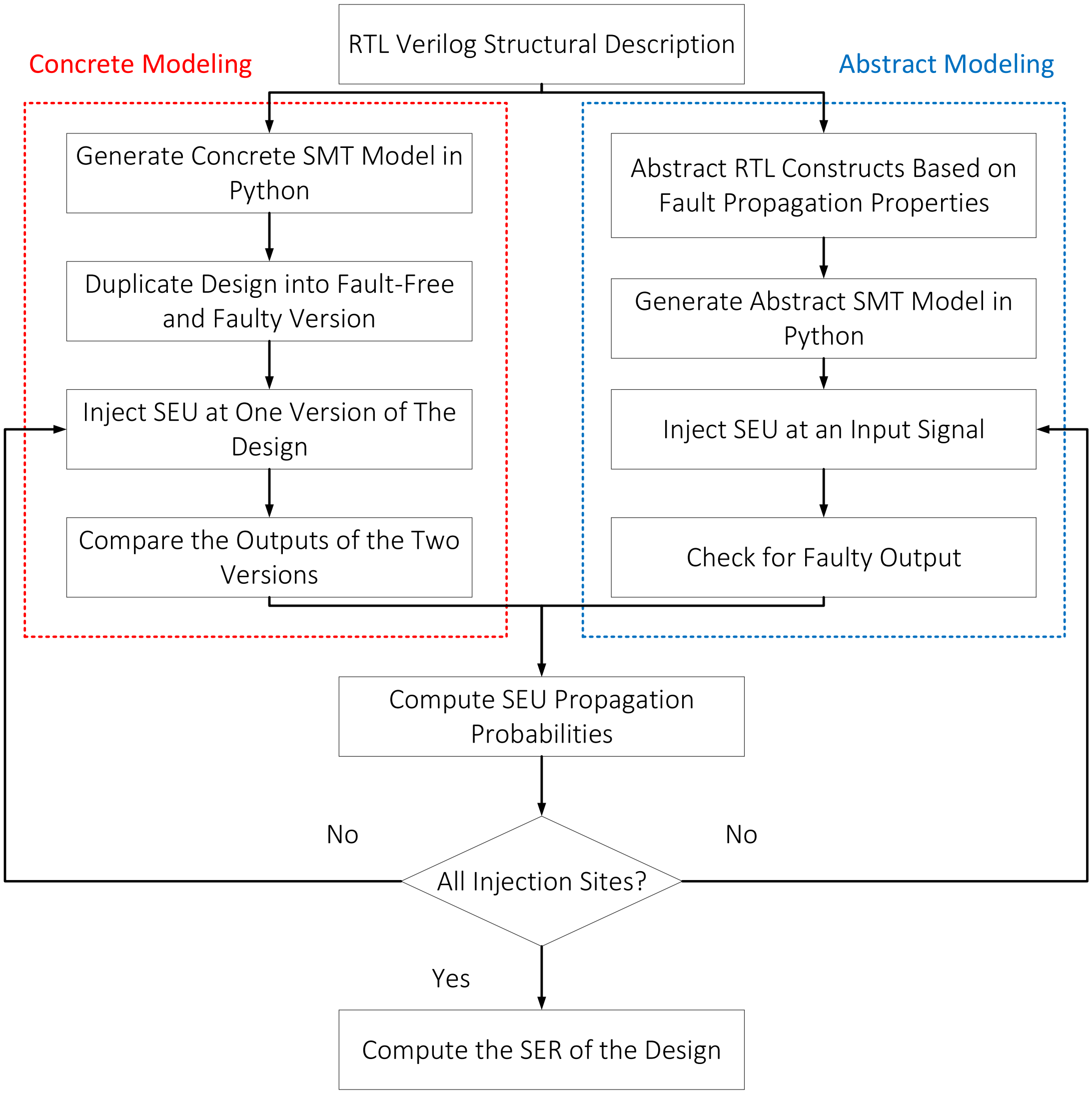

RTL Verilog Structural Description
Concrete Modeling
Abstract Modeling
Generate Concrete SMT Model in Python
Duplicate Design into Fault-Free and Faulty Version
Inject SEU at One Version of The Design
Compare the Outputs of the Two Versions
Abstract RTL Constructs Based on Fault Propagation Properties
Generate Abstract SMT Model in Python
Inject SEU at an Input Signal
Check for Faulty Output
Compute SEU Propagation Probabilities
No
No
All Injection Sites?
Yes
Compute the SER of the Design

Figure 3.2: Main Steps of the Proposed Methodology

3.3.1 Concrete Modeling

The modeling and analysis starts with an RTL Verilog description of the designs under test.

Generate Concrete SMT Model in Python

The first step in our methodology for the concrete modeling approach, shown in Figure 3.2, is to convert the RTL Verilog description into a Python SMT model. The SMT library includes operators supporting the theory of bit-vectors, which allows us to model basic bit-vector arithmetic and bitwise operations. Therefore, all RTL bit-vector operations were modeled using the SMT library. The circuit under test will be modeled as a Python function and maps the RTL constructs to equivalent SMT operations. The input parameters of the Python function representing the circuit are the primary inputs of the circuit and the function will return the outputs as shown in Listing 3.2.

```
def circuit(PI):
    ...
    Circuit Logic
    ...
    return PO
```

Listing 3.2: Python Function Defining the Combinational Circuit

Duplicate design into fault-free and faulty version

The generated model is then duplicated into one version considered as the fault-free and the other as the faulty version as shown in Figure 3.3. The SEUs are injected at the inputs of the faulty version of the circuit. The inputs of the combinational circuits are assumed to be latched. The outputs of the faulty and fault-free versions of the combinational circuits are compared to check if the SEU caused an error at the output.

This is done in the SMT Python model by having two sets of inputs (i.e. **PI** and **PI_f**) and two sets of outputs (i.e. **PO** and **PO_f**) to represent the fault-free and faulty versions as shown in Listing 3.3.

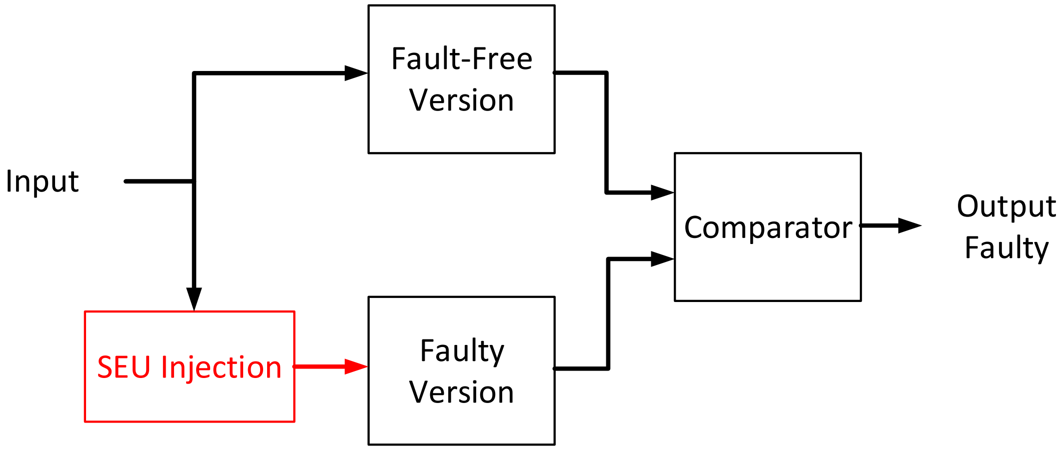


Figure 3.3: Proposed Modeling of SEU Propagation Using the Fault-Free and Faulty Versions Model

```
PO = circuit(PI)
PO_f = circuit(PI_f)
```

Listing 3.3: Python Functions of the Fault-Free and Faulty Versions

Inject SEU at one version of the design

The input vectors of both versions are asserted to be equal except for 1 bit. For a n-bit input vector A, all bits are asserted to be equal except for a bit k, where $k < n - 1$ as shown in Listing 3.4. The variable **formula** is a list containing all the assertions for a specific injection scenario. The next step is to add the assertions for fault propagation then check for the Satisfiability of the conjunction of all assertions added to the **formula**.

```
def injectSEU(k):
    formula = []
    for i in range(n):
        if(i == k):
            formula.append(Extract(i,i,PI) != Extract(i,i,PI_f))
        else:
            formula.append(Extract(i,i,PI) == Extract(i,i,PI_f))
    return formula
```

Listing 3.4: Fault Injection Function

Compare the outputs of the two versions

In [1], the fault injection and output comparison mechanisms are added to the design in the RTL description of the design and then the whole design is converted into a SAT instance. The outputs of the fault-free design and the faulty version are compared using an **XOR** operation. Every output bit of the fault-free version is **XOR**ed with its corresponding output bit in the faulty version. For each **XOR** operation, if the output is a `"1"`, it means that the SEU propagated to this specific output bit. The outputs of the **XOR** operations are then **OR**ed together to generate one output. This output bit is a `"1"` if the injected SEU propagated to at least one output, otherwise it is a `"0"`. The RTL Verilog was then converted into SMV format and then the SMV format was converted to CNF which is then sent to the SAT Solver.

In our modeling, the RTL Verilog designs are directly converted into SMT format without intermediate conversion steps. Therefore, SEU propagation is analyzed by simply verifying that the outputs of the faulty and fault-free versions are not equal as shown in Listing 3.5.

```
formula.append(PO != PO_f)
```

Listing 3.5: Fault Propagation Assertion

In this assertion, we verify if under any input condition the injected SEU will lead to an error at the output, i.e., the output of the faulty version is not equal to the output of the fault-free version.

3.3.2 Abstract Modeling

In this approach, we adapt the data type reduction technique used in [2]. The purpose of this approach is to investigate the effect and the applicability of a data type reduction technique on the modeling proposed in Section 3.3.1. The main steps of this modeling technique are shown in Figure 3.2.

The first steps starts by abstracting all RTL signals and operations based solely on fault propagation properties, i.e., the correct functionality of the circuit is not preserved. The multi-bit vectors are reduced to smaller bit-vectors. This results in a much smaller circuit and a smaller input vector search space.

RTL assignments, such as in Listing 3.1, are reduced to 1-bit operations. This is equivalent to a transparent channels, i.e., a faulty **start_addr** will result in a faulty **next_addr** signal. Since bit-level information is not available using the data type reduction technique, this directly implies faulty **addr_msb** and **addr_lsb** signals.

3.3.3 Estimate SER due to SEUs

In order for an SEU to result in a soft error, it must reach the primary outputs of the design. In other words, an active path must exist between the node or register where the SEU originates and the primary outputs of the circuit. If no active path exists, an SEU is said to be logically masked by the logical operation on its propagation path.

The probability that the injected SEU propagates to the output is computed as shown in equation (1). This probability is computed by dividing the total number of satisfying assignments (i.e., *Num_Assignments*) over the reduced randomized search space N of the total number of possible input vectors.

$$P(SEU_i) = \frac{Num\_Assignments_i}{N} \tag{1}$$

The Soft Error Rate (SER) is a rate used to classify the vulnerability of digital circuits to Soft Errors. In this specific case, we define the SER as being the summation of the propagation probabilities of all injections scenarios to any output.

Therefore, the Soft Error Rate (SER) of a design, which has m fault injection sites, is calculated as follows:

$$SER = \sum_{i=0}^{m-1} P(SEU_i) \tag{2}$$

3.3.4 SMT Model Count

In equation (1), *Num_ Assignments*, represents all satisfiable assignment for a given formula representing an SEU injection scenario. This number is computed iteratively by generating all solutions as shown in Listing 3.6. After generating the formula that includes the fault injection and propagation assertions, the total model count is computed and divided over the total search space to compute the vulnerability (denoted as vuln).

In this function we generate all satisfiable assignments using Z3's Python API. In order to do so, we generate a satisfiable assignment for the current injection scenario, and then add a new constraint that prevents the previous model from being generated again. This is repeated until the formula becomes unsatisfiable. This algorithm is shown in Listing 3.6.

```python
def genModels(formula):

    s = Solver()
    s.add(formula)
    count = 0

    while(s.check() == sat):
        m = s.model()
        block = [x != m.eval(x) for x in variables]
        s.add(Or(block))
        count += 1

    return count
```

Listing 3.6: Function to Generate All Satisfiable Assignments of a Given Formula

3.3.5 Examples

As an example, the proposed methodology was implemented on the 4-bit magnitude comparator circuit shown in Figure 3.4 to illustrate its main steps and investigate the results.

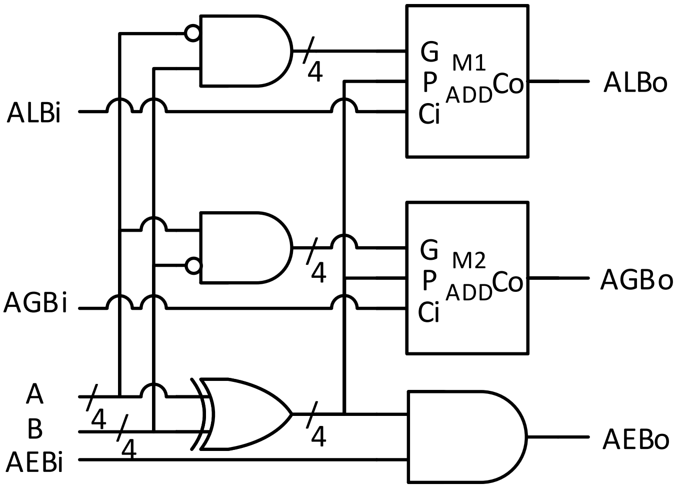


Figure 3.4: RTL Structure of the 74L85 Benchmark That is a 4-bit Magnitude Comparator

The comparator has 11 inputs and 3 outputs. The Verilog behavioral model of the circuit is shown in Listing 3.7.

```
module 74L85 (ALBi, AGBi, AEBi, A, B, ALBo, AGBo, AEBo);

    input[3:0]      A, B;
    input           ALBi, AGBi, AEBi;
    output          ALBo, AGBo, AEBo;
    wire[4:0]       CSL, CSG;

    assign CSL = ~A + B + ALBi;
    assign ALBo = ~CSL[4];
    assign CSG = A + ~B + AGBi;
    assign AGBo = ~CSG[4];
    assign AEBo = ((A == B) && AEBi);

endmodule
```

Listing 3.7: RTL Verilog Description of the 74L85 4-bit Magnitude Comparator Circuit

The RTL Verilog description of the module is then translated into an equivalent SMT model. Using Microsoft Z3's Python API, the description of the circuit is defined within a Python function as shown in Listing 3.8. The function's input parameters are the primary inputs of the designs and it returns the outputs as a tuple, which is a sequence of objects in Python.

```
def circuit(ALBi, AGBi, AEBi, A, B):

    CSL = ZeroExt(1,A) + ZeroExt(1,B) + ZeroExt(4,ALBi)
    ALBo = ~Extract(4,4,CSL)
    CSG = ZeroExt(1,A) + ZeroExt(1,~B) + ZeroExt(4,AGBi)
    AGBo = ~Extract(4,4,CSG)
    AEBo = BVRedAnd(~(A ^ B)) & AEBi

    return ALBo, AGBo, AEBo
```

Listing 3.8: Python Function Definition of the 74L85 4-bit Magnitude Comparator Circuit

In the main part of the program we create SMT bit-vector variables defining the inputs of the design for both the faulty and fault-free versions. Both sets of inputs are used as inputs to the function defining the circuit shown in Listing 3.8. The tuples returned are then unpacked into the outputs of the fault-free and faulty versions. The main part of the program is shown in Listing 3.9.

```
A = BitVec('A', 4)
B = BitVec('B', 4)
ALBi = BitVec('ALBi', 1)
AGBi = BitVec('AGBi', 1)
AEBi = BitVec('AEBi', 1)

A_f = BitVec('A_f', 4)
B_f = BitVec('B_f', 4)
ALBi_f = BitVec('ALBi_f', 1)
AGBi_f = BitVec('AGBi_f', 1)
AEBi_f = BitVec('AEBi_f', 1)

ALBo, AGBo, AEBo = circuit(ALBi, AGBi, AEBi, A, B)
ALBo_f, AGBo_f, AEBo_f = circuit(ALBi_f, AGBi_f, AEBi_f, A_f, B_f)
```

Listing 3.9: Python Two-Versions Model of the 74L85 4-bit Magnitude Comparator Circuit

The fault injection and propagation assertions are then added to the formula as previously shown in Listing 3.4 and Listing 3.5. The propagation probabilities are then computed for every injection scenario. The results for the 74L85 are shown in Table 1.

Table 1: SEUs Propagation Probabilities for the 74L85 4-bit Magnitude Comparator Circuit

Primary Inputs (PIs)	**Propagation Probability** $P(SEU_i)$
A[3]	0.765625
A[2]	0.53125
A[1]	0.28125
A[0]	0.15625
B[3]	0.765625
B[2]	0.53125
B[1]	0.28125
B[0]	0.15625
ALBi	0.0625
AGBi	0.0625
AEBi	0.0625
Soft Error Rate (SER)	3.65625

It is possible to verify the computed propagation probabilities and the SER by analyzing the design structure. An SEU injected at input signal **AEBi** can only propagate to output **AEBo** under the condition that the input bit-vectors **A** and **B** are equal, i.e., $A == B$. There exists a total of 128 input vector combinations that satisfy this condition out of a total of 2^{11} which results in a propagation probability of 0.0625. When using concrete modeling, the computed SER is 3.65.

When using abstract modeling, we reduce the bit-vectors **A** and **B** to 1-bit signals and compute the propagation probabilities. It is assumed that all the bits of an n-bit signal have equal vulnerabilities, and that the vulnerability of each bit is equal to the computed vulnerability of the reduced bit-vector. This will results in a propagation probability of 1 for SEUs injected at inputs **A** and **B**. Assuming equal vulnerability for all the bits, this results in an SER of 10.5. Using abstract modeling in this cases results in a loss in accuracy of 187.2%. However, the computation time is 0.6 seconds while for concrete modeling the computation time is 17.55 seconds.

As another example, the proposed methodology was implemented on the 74283 Fast Adder circuit shown in Figure 3.5.

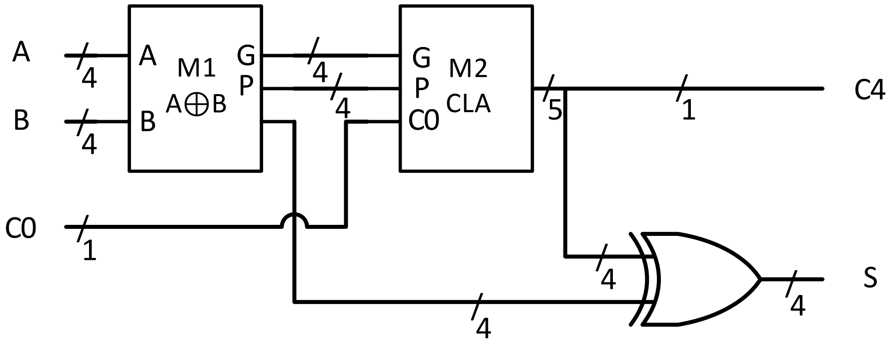


Figure 3.5: RTL Structure of the 74283 Benchmark that is a Fast Adder Circuit

This circuit has 9 inputs and 5 outputs. The Verilog behavioral model of the circuit is shown in Listing 3.10.

The RTL Verilog description of the module is then translated into an equivalent SMT model. Using Microsoft Z3's Python API, the description of the circuit is defined within a Python function as shown in Listing 3.11.

```
module 74283 (C0, A, B, S, C4);

    input[3:0]    A, B;
    input         C0;
    output[3:0]   S;
    output        C4;
    wire[4:0]     CS;

    assign CS = A + B + C0;
    assign S = CS[3:0];
    assign C4 = CS[4];

endmodule
```

Listing 3.10: RTL Verilog Description of the 74283 Fast Adder Circuit

```
def circuit(C0, A, B):

    CS = ZeroExt(1,A) + ZeroExt(1,B) + ZeroExt(4,C0)
    S = Extract(3,0,CS)
    C4 = Extract(4,4,CS)

return S, C4
```

Listing 3.11: Python Function Definition of the 74283 Fast Adder Circuit

The main part of the program is shown in Listing 3.12. The 74283 Fast Adder Circuit has two 4-bit input vectors **A** and **B**, and a 1-bit input **C0**. An SEU injected at those inputs will always propagate to one of the two outputs **S** and **C4**. This will result in a uniform propagation probability of 1 for all bits of all inputs, which results in an SER of 9.0.

When using abstract modeling, we reduce the input bit-vector signals **A** and **B** to 1-bit signals. When computing the propagation probabilities using the abstract model, we obtain the same value in SER of 9.0. The computation time for concrete modeling in this case is 6.56 seconds while for abstract modeling it is 0.14 seconds.

We used these two examples to provide a detailed step-by-step description of our methodology. Moreover, these two examples show that abstract modeling, which utilizes our proposed adaptation of a data type reduction technique, can sometimes

```
A = BitVec('A', 4)
B = BitVec('B', 4)
C0 = BitVec('C0', 1)

A_f = BitVec('A_f', 4)
B_f = BitVec('B_f', 4)
C0_f = BitVec('C0_f', 1)

S, C4 = circuit(C0, A, B)
S_f, C4_f = circuit(C0_f, A_f, B_f)
```

Listing 3.12: Python Two-Versions Model of the 74283 Fast Adder Circuit

result in great improvement in computation time with little or no loss in accuracy. However, in other cases it can result in significant loss in accuracy of the computed SER.

In the next section we apply our proposed approach on more ISCAS85 benchmarks to investigate its efficiency on larger designs.

3.4 Experimental Results

In this section, we report the experiments used to validate the proposed methodology and its efficiency. The proposed modeling and analysis are fully automated using the Microsoft Z3 SMT solver [12] and Python scripts. Our experiments were performed on a workstation with an Intel(R) Core(TM) i7-6820HQ running at 2.70 GHz and with 16 GB RAM.

3.4.1 Accuracy Analysis

The goal of this analysis is to evaluate the loss in accuracy when the abstract modeling technique is used. For this analysis, different ISCAS85 benchmarks [13] and MSI components in the 74xxx series were analyzed. In order to compare the accuracy, the same designs were modeled, analyzed, and their SERs were estimated under both modeling techniques. The results of this analysis are reported in Table 2. The second column shows the number of primary inputs of the design. The third column shows the number of input multi-bit signals. For example, the MSI component 74283 has three input signals: two 4-bit input signals and one 1-bit input signal, that is, 9 primary inputs. The computed SERs based on concrete modeling and abstract modeling are presented in columns 4 and 5 of Table 2 respectively. The percentage difference in the computed SERs is presented in column 6 of Table 2.

Table 2: Comparison Between the SER of the Concrete Modeling Technique and the Abstract Modeling Technique

Circuit	**PIs**	**Signals**	**Concrete SER**	**Abstract SER**	**Percentage Difference**
74283	9	3	9	9	0
74182	9	3	4.57	8.25	80.36
74181	14	5	7.02	5.37	23.44
74L85	11	5	3.65	10.5	187.67
c432	36	4	6.88	8.46	22.97
c499	41	3	32.00	32.01	0.03

It is observed that in some cases the difference in the computed SER varies greatly when the abstract modeling is used, depending on the analyzed design. For example, the difference in the computed SER can be as high as 187.67% in the case of the MSI component 74L85. On the other hand, the difference can be as low as 0 for the case of the 74283 circuit.

This difference in the estimated SERs between the concrete and the abstract modeling techniques can be explained by the following reasons:

- The concrete modeling allows the injection of SEUs at every input bit of the design. Therefore, the vulnerability of every bit is obtained and summed up to obtain the exact SER of the whole design. However, with the abstract modeling, we do not have access to the affected bits and therefore, the whole vector is assumed to be faulty. This assumption will lead to an over approximation in the SER in most cases since all the bits are treated as equally critical.

- With the concrete modeling, the propagation of the SEUs can be accurately traced. However, when the abstract modeling is used, we can only keep track of the fault state of the whole signal. Therefore, tracking faults at every bit becomes impossible since the correct functionality of the circuit is lost and only error propagation properties at the level of signals is available.
- With the concrete modeling, it is possible to exhaustively analyze the input vector search space for each injection scenario, to obtain an accurate SER value. However, with the abstract modeling such analysis is not possible and only an over approximation or an under approximation of the percentage of input vectors that affects the SEU propagation is generated.

Next, we investigate how much the performance can be improved at a cost in accuracy when using abstract modeling instead of concrete modeling.

3.4.2 Performance Analysis

The second analysis compares the performance in terms of computation time for both the abstract and the concrete modeling. Although it is known that abstraction reduces computation time, the goal here is to investigate the trade-off between the loss in accuracy and the gained speed-up in analysis time. Table 3 shows the computation times for analyzing the same circuits using the two modeling approaches. The fourth and fifth column represent the computation times of the concrete and abstract modeling, respectively. The average speed-up of the abstract modeling over the concrete modeling for all circuits analyzed is around 67.33.

Table 3: Comparison Between the Computation Time for the Concrete Modeling Technique and the Abstract Modeling Technique

Circuit	**PIs**	**Signals**	**Concrete** **CPU Time (sec)**	**Abstract** **CPU Time (sec)**	**Speed** **Up**
74283	9	3	6.56	0.14	46.86
74182	9	3	3.14	0.18	17.06
74181	14	5	370.65	2.29	162.21
74L85	11	5	17.5	0.6	29.17
c432	36	4	505.25	3.54	142.50
c499	41	3	92.94	14.98	6.20
Average	-	-	166.01	3.62	67.33

Table 4 represents the ratio of the loss in accuracy in the estimated SER over the gain in the analysis time. A small ratio implies a better trade-off, i.e., a small loss in SER accuracy over a large speed-up of the analysis time.

Based on the results, it can be observed that the abstract modeling has a different impact based on the design behavior. For example, the best trade-off was observed in the case of the 74283 and the c499. For the c499 design, the speed-up is around 6 while the percentage of the loss in accuracy is only 0.03%. This is due to the fact that, all the bits in every input signal have equal vulnerabilities. However, for the case of the 74L85 benchmark, the speed-up is around 29 while the percentage of the loss in accuracy is around 187%. This can be partially explained by the fact that this design is a comparator, i.e., with the abstract modeling many of the injected SEUs are considered to propagate to the output while they should have been logically masked.

Table 4: The Ratio of the Speed-Up in CPU Time over the Percentage Difference in Computed SER

Circuit	**PIs**	**Signals**	**Percentage Difference in SER**	**Speed Up in CPU Time**	**Ratio**
74283	9	3	0	46.86	0
74182	9	3	80.36	17.06	4.71
74181	14	5	23.44	162.21	0.144
74L85	11	5	187.67	29.17	6.43
c432	36	4	22.97	142.50	0.161
c499	41	3	0.031	6.20	0.005

3.5 Summary

In this chapter, we proposed a new methodology to investigate the vulnerability of combinational designs at RTL due to SEUs. The SMT theories of bit-vectors are utilized. Two modeling techniques were used in order to evaluate the efficiency of SMT modeling for SEU propagation: concrete modeling and abstract modeling. Concrete modeling preserves the functionality of the design. Abstract modeling uses data type reduction to reduce complex bit-vector operations to simpler Boolean operations. In the experiments, the two modeling approaches are compared. Our results show that based on the design behavior, abstract modeling can be used to generate acceptable estimates of the SER in a shorter time. For example, for the c499, an SER with 0.03% inaccuracy can be generated with a computation time that is 6 times less than concrete modeling. Later in this work, we propose a different modeling method to improve the efficiency of our computation. Moreover, in order to handle larger designs, we will use an approximate model counting approach. In the next section, we extend the modeling approach proposed in this chapter to handle sequential circuits.

Chapter 4

SMT Modeling and Analysis of SEUs Propagation in Sequential Circuits

4.1 Introduction

In this chapter, we introduce a new methodology to estimate the vulnerability of sequential circuits to SEUs at RTL. This method is applicable on RTL word-level designs, without requiring synthesis or conversion to pure Boolean logic. This chapter introduces a new modeling of SEUs propagation as a Satisfiability problem using Satisfiability Modulo Theories (SMTs) similar to what is proposed in chapter 3. The model is extended to handle SEU propagation through state elements. The basic RTL operations (e.g. logical operators, reduction operators, arithmetic operators, and conditional statements) are modeled in the presence of SEUs. The proposed methodology allows the analysis of SEUs propagation from each vulnerable node to the output of subsequent cycles of the sequential circuit. Moreover, the proposed modeling and analysis provides an early estimate of the Soft Error Rate (SER) of the design based on an approximate model counting approach.

The rest of this chapter is organized as follows. In Section 4.2 we discuss approximate model counting. In Section 4.3 we present our proposed SMT modeling for sequential circuits. In Section 4.4 we explain our methodology for computing the

SER. In Section 4.6, we explain our experiments and results. Section 4.7 concludes this work.

4.2 Approximate Model Counting

If the constructed SMT model satisfies the set of assertions being verified, the SMT solver generates a satisfiability assignment. In such assignment, each variable in the model is assigned a valid value such that the SMT model satisfies the verified assertions. However, when dealing with real systems with a large number of variables, model counting (#SAT), i.e., counting the number of satisfying assignments of a first order logic formula, is a problem of significant theoretical and practical complexity. Different methods to reduce the complexity of this problem and to eliminate the need for exact model counting were proposed. One of the main approaches in this area is approximate model counting. This is used in tools such as ApproxCount [41], SearchTreeSampler [42], SE [43], SampleSearch [44].

Recently, a new technique that is based on approximate model counting was proposed. This technique was successfully implemented in a tool called SMTApproxMC [45]. The accuracy and the scalability of this tool were demonstrated through different experiments. Results show that SMTApproxMC [45] scales to formulas with tens of thousands of variables. Moreover, based on the desired level of confidence, SMTApproxMC [45] can provide bounds that are close to the exact model count. Later in this chapter, we will explain the utilization of this tool in our method to efficiently provide more accurate estimations of the soft error rate due to SEUs.

4.3 SMT Modeling of Sequential Circuits

In this section, we explain the proposed modeling of SEU propagation at RTL. First we explain proposed SMT modeling of the combinational part of the sequential circuit. Next, we explain the unrolling of the sequential circuits. The main steps of the proposed methodology are shown in Figure 4.1.

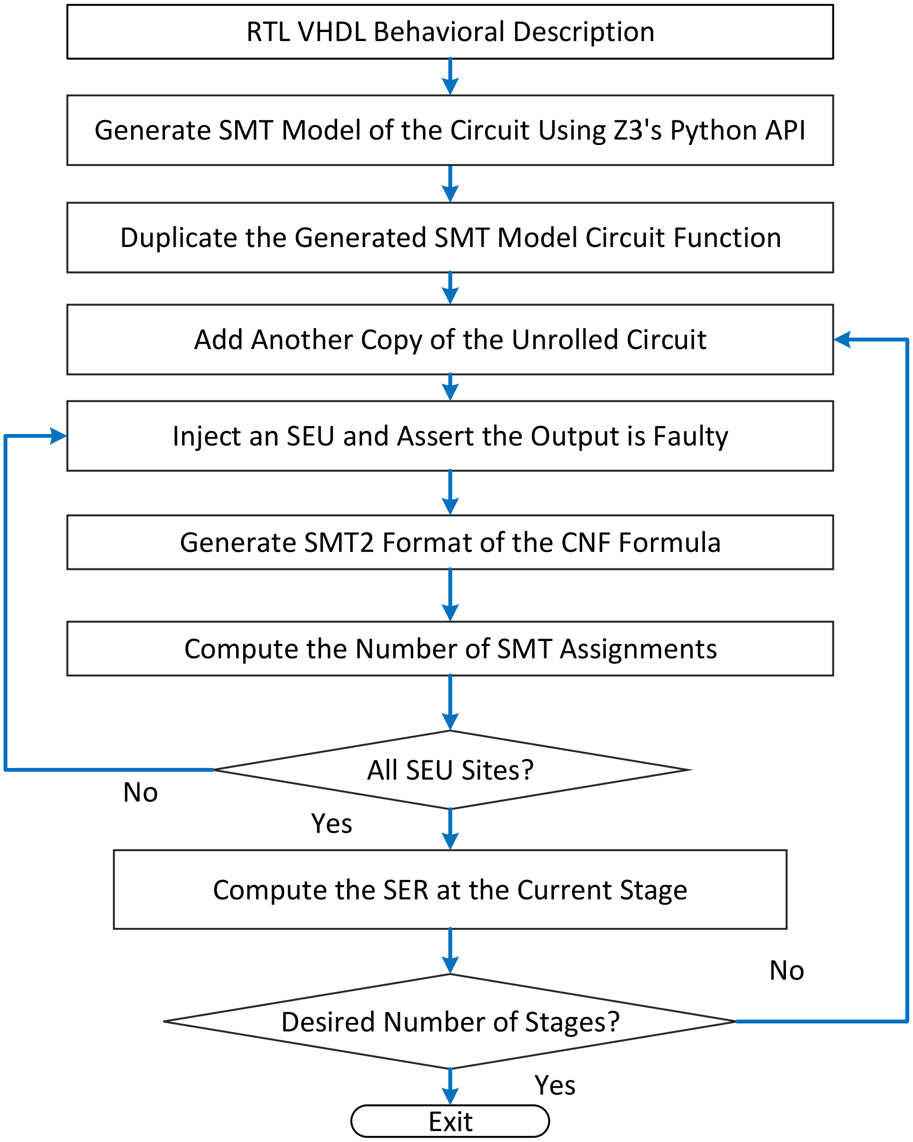

RTL VHDL Behavioral Description
Generate SMT Model of the Circuit Using Z3's Python API
Duplicate the Generated SMT Model Circuit Function
Add Another Copy of the Unrolled Circuit
Inject an SEU and Assert the Output is Faulty
Generate SMT2 Format of the CNF Formula
Compute the Number of SMT Assignments
All SEU Sites?
No
Yes
Compute the SER at the Current Stage
No
Desired Number of Stages?
Yes
Exit

Figure 4.1: Main Steps of the Proposed Methodology for the Analysis of SEU Propagation in Sequential Circuits

4.3.1 SMT Modeling of the Combinational Part

The proposed methodology starts with a fully synthesizable RTL VHDL description of the sequential circuit. The VHDL description is converted into an SMT formula.

Thereafter, two copies of the generated SMT formula are made. Similarly to chapter 3, these two duplicates will represent a fault-free version and a faulty version of the design, where the SEU will be injected at the state elements.

The SMT theory of bit-vectors and linear arithmetic allow us to model the various RTL constructs directly into an SMT formula.

If Statements and Case Statements

For example, the *if* statements can be modeled using SMT *ITE* (*If-Then-Else*) statements. The *ITE* statement has the following format: **If** "*statement*" **then** "*statement*" **else** "*statement*". For example, for a generic *if statement* such as the one shown in Listing 4.1, it can be modeled as two nested *ITE* statements as shown in Listing 4.2. Note that in Microsoft Z3's Python API, the *ITE* statement function is called *If* with a capital I to differentiate from the regular Python *if* statement.

Arithmetic Operators

SMT supports arithmetic operation on bit-vectors and integer variables. This allows arithmetic operations such as addition, subtraction, multiplication and division to be directly modeled as SMT operations For example, the assignment $F <= A + B$ in Listing 4.1 is assigned in the nested *ITE* statement of Listing 4.2 as a bit-vector addition in SMT.

Concatenate and Slice Operators

SMT supports bit-vector concatenation and extraction. For example, in Listing 4.1, the assignment $F <= C(3\ \mathbf{downto}\ 0)$ can be converted into SMT format using the *Extract* SMT function as shown in Listing 4.2. Similarly, the bit-vector concatenation $F <= D\ \&\ E$ can be performed in SMT format using the *Concat* SMT function as shown in Listing 4.2.

```
if (A = 1) and (B = 2) then
    F <= A + B;
elsif (A >= 5) then
    F <= C(3 downto 0);
else
    F <= D & E;
end if;
```

Listing 4.1: VHDL If Statement

```
F = If(A == 1 && B == 2, A + B,
        If(A >= 5, Extract(3,0,C),
            Concat(D,E)))
```

Listing 4.2: SMT Modeling of Different RTL Constructs

4.3.2 Unrolling of the Sequential Circuit

In order to investigate the propagation of SEUs in a sequential circuit, the combinational part must be unrolled to simulate state transitions. The goal of this unrolling approach is to investigate SEU propagation through different states.

The circuit unrolling approach is used in existing sequential analysis techniques, e.g., [46] and [47]. Figure 4.2 shows the result of unrolling both the faulty and fault-free version of the sequential circuit for analyzing SEU propagation. The steps of this unrolling approach are the following:

1. The sequential design is unrolled by making k number of copies of its combinational part. Registers are replaced with wires and each copy (or stage) represents a state of the design.

2. Two copies of the unrolled circuit are generated, one to be used as faulty and the other one as fault-free.

3. An SEU will be injected at one of the state lines in the first stage of the faulty copy of the unrolled circuit. Thereafter, the outputs of each stage of the faulty and the error-free version of the design are compared for each fault injection as shown in Figure 4.2.

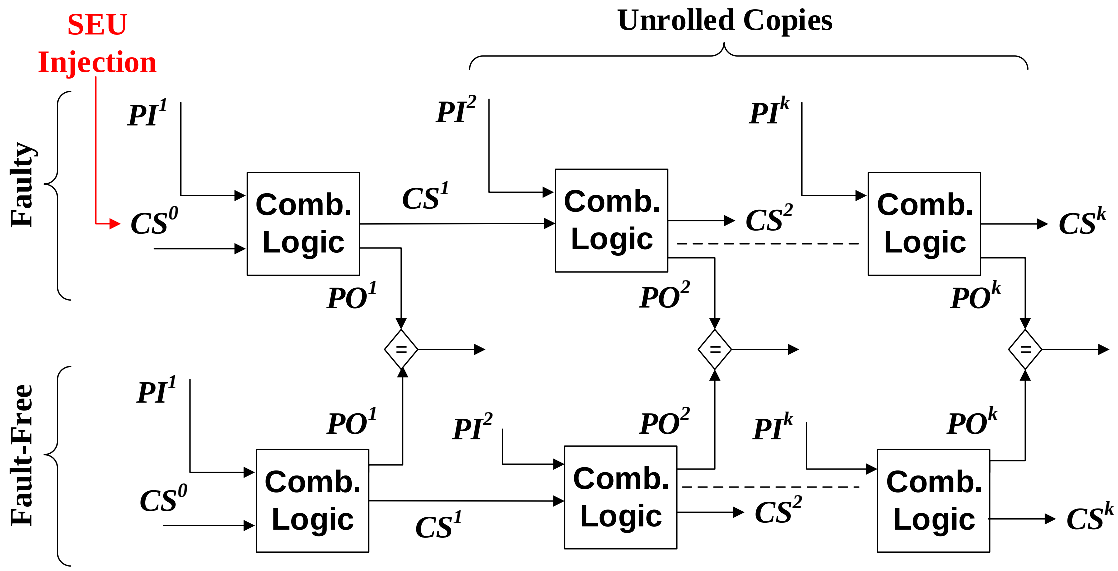


Figure 4.2: Methodology for Fault Propagation Analysis in Sequential Circuits Using Unrolling

```
def circuit(PI, CSprev):

    ...
    Circuit Logic
    ...

    return PO, CSnext
```

Listing 4.3: Python Function Defining the Circuit

Using Microsoft's Z3 [12] Python API, the functions which are modeling the combinational part are defined using SMT constructs as shown in Listing 4.3. This is similar to how the circuit is defined in Python as a function as described in chapter 3. However, the difference when modeling sequential circuits is that the function has an additional input parameter and returns an extra value. The function's input parameters are the primary inputs and the previous state, i.e., **PI** and **CSprev** respectively. The outputs of the function are the primary outputs and the next state, i.e., **PO** and **CSnext** respectively.

```
PO1, CS1 = circuit(PI0, CS0)
PO2, CS2 = circuit(PI1, CS1)
 ...
POn, CSn = circuit(PIn, CSn)

PO1_f, CS1_f = circuit(PI0, CS0_f)
PO2_f, CS2_f = circuit(PI1, CS1_f)
 ...
POn_f, CSn_f = circuit(PIn, CSn_f)
```

Listing 4.4: SMT Circuit Unrolling

The unrolling is done as shown in Listing 4.4. The primary inputs **PI0**, **PI1**, ..., **PIn** are defined as SMT variables. The initial state, **CS0**, is assumed to be free, therefore it is also defined as a variable. As shown in Figure 4.2, the output state of the first copy of the circuit, i.e. CS1, is used as an input of the next copy until we have unrolled the circuit n times.

The chain of unrolled copies is also duplicated to represent the fault-free and faulty versions of the circuits. At every stage, the fault-free outputs **PO1**, **PO2**, ..., **POn** are compared to the faulty versions **PO1_f**, **PO2_f**, ..., **POn_f** to check if the injected SEU at the initial stage (**CS0**) has reached an output. Based on design criticality and behavior, it is possible to define a threshold at which the analysis stops and no further unrolled copies are required. As discussed in [47], unrolling the circuit two times leads to negligible approximation error.

4.4 SEU Propagation and SER Estimation

In this chapter, we use a different approach to compute the Soft Error Rate (SER). First we compute the propagation probabilities from every input bit to every output bit, and not to the output as a whole. This provides more details on the vulnerability of a circuit. Moreover, we compute the SER for the outputs at every cycle following the injection cycle of the SEU. In this section, the proposed analysis of SEU propagation is explained in detail. First we explain how SEUs are injected into our proposed

SMT model. Thereafter, we explain how the SER is computed.

4.4.1 Soft Error Rate Calculation

We define the SER as the summation of the vulnerability of each output at every stage. Therefore, an SER is generated for every stage following the injection stage (i.e. stage 0). We define the vulnerability of every output as the summation of the propagation probabilities of every injected SEU. The probability of propagation of an SEU injected at register bit i to an output j at stage k is given by equation (3). It is the total number of satisfiable assignments ($Num\_Assignments$), i.e. initial state and input vectors that allow the SEU to propagate to the output over the size of the search space (N).

$$P(SEU_{i \longrightarrow j,k}) = \frac{Num\_Assignments_{i \longrightarrow j,k}}{N} \tag{3}$$

Therefore, the vulnerability of an output j at stage k for a circuit which has m vulnerable register bits, is calculated as shown in equation (4).

$$Vuln(j,k) = \sum_{i=0}^{m-1} P(SEU_{i \longrightarrow j,k}) \tag{4}$$

Finally, the SER of a sequential circuit at stage k, which has n outputs, is calculated as shown in equation (5).

$$SER(k) = \sum_{j=0}^{n-1} Vuln(j,k) \tag{5}$$

4.4.2 SEU Injection and Propagation

In order to compute the SER, SEU injection and propagation assertions are added to the model presented in Section 4.3. To inject faults and check for propagation to a specific output bit, the initial register state and the unrolled circuits shown in Listing 4.4 are duplicated, into fault-free and faulty versions. The outputs are then compared at every stage.

The function defined to inject an SEU at bit *i*, for an initial state register variables called **regs_0** is defined in Listing 4.5. The formula will be asserted when checking for satisfiability of the fault propagation scenario. In this function, we add a list of assertions to the original formula. This function asserts that the initial states of the faulty and fault-free versions are equal, except for one bit *i*.

```
def injectSEU(i):
    formula = []
    for j in range(regLength):
        if(j == i):
            formula.append(Extract(j,j,CS0) != Extract(j,j,CS0_f))
        else:
            formula.append(Extract(j,j,CS0) == Extract(j,j,CS0_f))
    return formula
```

Listing 4.5: Function to Inject SEU

In order to check if an SEU was able to propagate to the output at stage *k*, we must assert that the output of the fault-free copy is not equal to the output of the faulty copy, i.e. **POk** != **POk_f**.

4.4.3 Soft Error Rate Computation

In order to compute the vulnerability of every output and the final SER, we compute the propagation probability for every possible SEU injection site to every output. For every stage of the unrolling the function computeSER is called. This function takes three inputs: **i**, **variables**, and **vulnMatrix**. The variable **i** represents the stage for which we are computing the SER.

The variable **variables** is a list of the variables of the current SMT formula. For example, at stage 0, the variables of the SMT formula will be the initial register state (i.e. **CS0**) and the primary inputs of stage 0 (i.e. **PI0**). For stage 1, the variables will be the initial register state (i.e. **CS0**), the primary inputs for stage 0 (i.e. **PI0**) as well as the primary inputs of stage 1 (i.e. **PI1**). For example in Figure 4.2, we see that the output **PO1** depends only on **PI1** and **CS0**, while **PO2** depends on

CS0, **PI1** and **PI2** etc. These variables are required to compute the propagation probability since they represent the search space. The size of the search space for stage k is $2^{CS0+k*PI}$.

The function **computeSER** in Listing 4.6 loops through every register bit and injects a fault at bit i, then for every output j, it asserts that bit j of the fault-free output is not equal to bit j of the faulty output. The vulnerability is then computed using the function **genModels**, which takes as input the formula and the list of variables. This probability is then stored in a matrix (**vulnMatrix**). The SER at any given stage, is simply the sum of vulnerabilities. In Listing 4.6, after generating the formula that includes the fault injection and propagation assertions, the total model count is computed using the function **genModels** and divided over the total search space to compute the vulnerability (denoted as **vuln**).

```
def computeSER(stage,variables,vulnMatrix):
SER = 0
for i in range(regLength):

    for j in range(outLength):

        formula = injectSEU(i)
        formula.append(Extract(j,j,PO[stage])
                     != Extract(j,j,PO_f[stage]))
        vuln = genModels(formula, variables)
                     / 2**numOfBits(variables)
        vulnMatrix[stage][i][j] = vuln
        SER += vuln

return SER
```

Listing 4.6: Function to Compute the SER

4.4.4 SMT Model Count

In Listing 4.6, after generating the formula that includes the fault injection and propagation assertions, the total model count is computed using the function **genModels** and divided over the total search space to compute the vulnerability (denoted as

vuln). In our analysis we performed this step in two ways and compared the results.

Exhaustive Method

In this method we generate all satisfiable assignments using Z3's Python API. In order to do so, we generate a satisfiable assignment for the current injection scenario, and then add a new constraint that prevents the previous model from being generated again. This is repeated until the formula becomes unsatisfiable. This algorithm is shown in Listing 4.7.

```
def genModels(formula):

    s = Solver()
    s.add(formula)
    count = 0
    while(s.check() == sat):
        m = s.model()
        block = [x != m.eval(x) for x in variables]
        s.add(Or(block))
        count += 1

return count
```

Listing 4.7: Function to Generate All Satisfiable Assignments of a Given Formula

Approximate Method

Exhaustive model generation becomes unusable when unrolling for several cycles. The number of variables increases linearly thus increasing the search space exponentially. For this reason, we used the SMTApproxMC tool [45] discussed in Section 4.2 to perform the same analyses on different circuits.

The results obtained from the exhaustive method and the approximate method will be used to investigate the trade-off in accuracy and performance.

4.5 Example

In order to illustrate the methodology described in the previous subsection, we analyze the vulnerability of a simple 4-bit unsigned up counter with synchronous reset. The VHDL RTL description of the circuit is shown in Listing 4.8.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counter is
    port(clk, reset, en : in std_logic;
        cout : out std_logic_vector(3 downto 0));
end counter;

architecture rtl of counter is
signal count: std_logic_vector(3 downto 0);
begin

    process (clk, load)
    begin
        if (clk'event and clk='1') then
        if (reset = '1') then
                count <= "0000";
            elsif (en = '1') then
                count <= count + 1;
            end if;
        end if;
    end process;

cout <= count;
end rtl;
```

Listing 4.8: VHDL RTL Description of a 4-bit Unsigned Up Counter with Synchronous Reset

When unrolling a circuit multiple times, the search space becomes exponentially large, so for this reason we chose a small circuit to illustrate the procedure in our example. The circuit has three primary inputs, **load**, **en** and **cin**. The **cin** signal is a 4-bit input bit-vector of the value to be loaded into the counter registers. The **load** signal is used as a synchronous data from the primary input **cin** into the registers.

The **en** is used as an enable signal for the up counter, i.e., the counter circuit will count up only if **en** is set to high.

As discussed earlier, Z3 supports boolean, bit-vector, linear arithmetic and ITE (if-then-else) operators which greatly facilitates the modeling of VHDL behavioral descriptions. Most sequential designs contain several *if-then-else* statements and *case* statements. The 4-bit counter in Listing 4.8 can be modeled in SMT using one ITE statement as shown in Listing 4.9. The ITE statement in Microsoft Z3's Python API is written as If with a capital I. We concatenate the primary inputs into one vector **PI** and the registers of the previous state into one vector **CSprev** and extract the appropriate bits inside the function defining the combinational part of the circuits. This is done in order to keep the code in the main function of the program unchanged and only require modifying the function defining the circuit.

```
def circuit(PIs, CSprev):
    reset = Extract(0,0,PIs)
    en = Extract(1,1,PIs)

    count = CSprev

    count_next = If(reset == 1, BitVecVal(0,4),
                    If(en == 1,count + 1,
                       count))

    POs = count
    CSnext = count_next

    return POs, CSnext
```

Listing 4.9: Python SMT Description of a 4-bit Unsigned Up Counter with Synchronous Reset

Since the design will be unrolled to represent the state transitions, note that the 4-bit register count, which represents the current state is an input to the SMT function representing the sequential design. The next state of count, called **count_next** is then evaluated using an ITE statement and returned as an output.

SEUs are then injected at the state elements, i.e. at every bit in the registers.

The SEU propagation probability is then computed from every injection site to every output of the design for the desired number of subsequent clock cycles. In Table 5 and Table 6, we present the results of this analysis for the 4-bit Unsigned Up Counter with Synchronous Reset. The columns represent the output bits at every stage and the rows represent the bits of the initial stage register. As discussed earlier, the initial stage represents the stage at which the SEU is injected.

From the description of the circuit, it can be seen that an SEU injected in any of the bits of the register **count** will only be visible at that same bit at the output **cout** of the current cycle. The SER for stage 0 can be calculated to be 4.0 as shown in Table 5. For stage 1, an SEU injected at bit 0 of the register **count** will be masked only if the **reset** signal is asserted, i.e. its propagation probability is 0.5. In Table 5 and Table 6, we show the propagation probabilities and the SERs for 5 clock cycles following SEU injection.

Table 5: SER of 4-bit Unsigned Up Counter with Synchronous Reset for Stage 0 and Stage 1

Stages	**Stage 0**				**Stage 1**			
Regs	**cout[3]**	**cout[2]**	**cout[1]**	**cout[0]**	**cout[3]**	**cout[2]**	**cout[1]**	**cout[0]**
count[3]	1.0	0.0	0.0	0.0	0.5	0.0	0.0	0.0
count[2]	0.0	1.0	0.0	0.0	0.0625	0.5	0.0	0.0
count[1]	0.0	0.0	1.0	0.0	0.0625	0.125	0.5	0.0
count[0]	0.0	0.0	0.0	1.0	0.0625	0.125	0.25	0.5
Vul	1.0	1.0	1.0	1.0	0.6875	0.75	0.75	0.5
SER	4.0				2.6875			

The advantage of computing the SER and the specific register vulnerabilities using

Table 6: SER of 4-bit Unsigned Up Counter with Synchronous Reset for Stage 2 and Stage 3

Stages	**Stage 2**				**Stage 3**			
Regs	**cout[3]**	**cout[2]**	**cout[1]**	**cout[0]**	**cout[3]**	**cout[2]**	**cout[1]**	**cout[0]**
count[3]	0.25	0.0	0.0	0.0	0.125	0.0	0.0	0.0
count[2]	0.0625	0.25	0.0	0.0	0.046875	0.125	0.0	0.0
count[1]	0.0625	0.125	0.25	0.0	0.0390625	0.078125	0.125	0.0
count[0]	0.03125	0.0625	0.125	0.25	0.015625	0.03125	0.0625	0.125
Vul	1.0	1.0	1.0	1.0	0.6875	0.75	0.75	0.5
SER	1.46875				0.7734375			

this approach, is that it provides detailed values for all the possible propagation paths from injections sites to specific output bits. This information can be used for designing more reliable circuits and applying fault mitigation techniques on the most critical paths.

In the following section, we conduct experiments on larger sequential circuits in order to investigate the efficiency of approximate model counting performed by the SMTApproxMC tool [45].

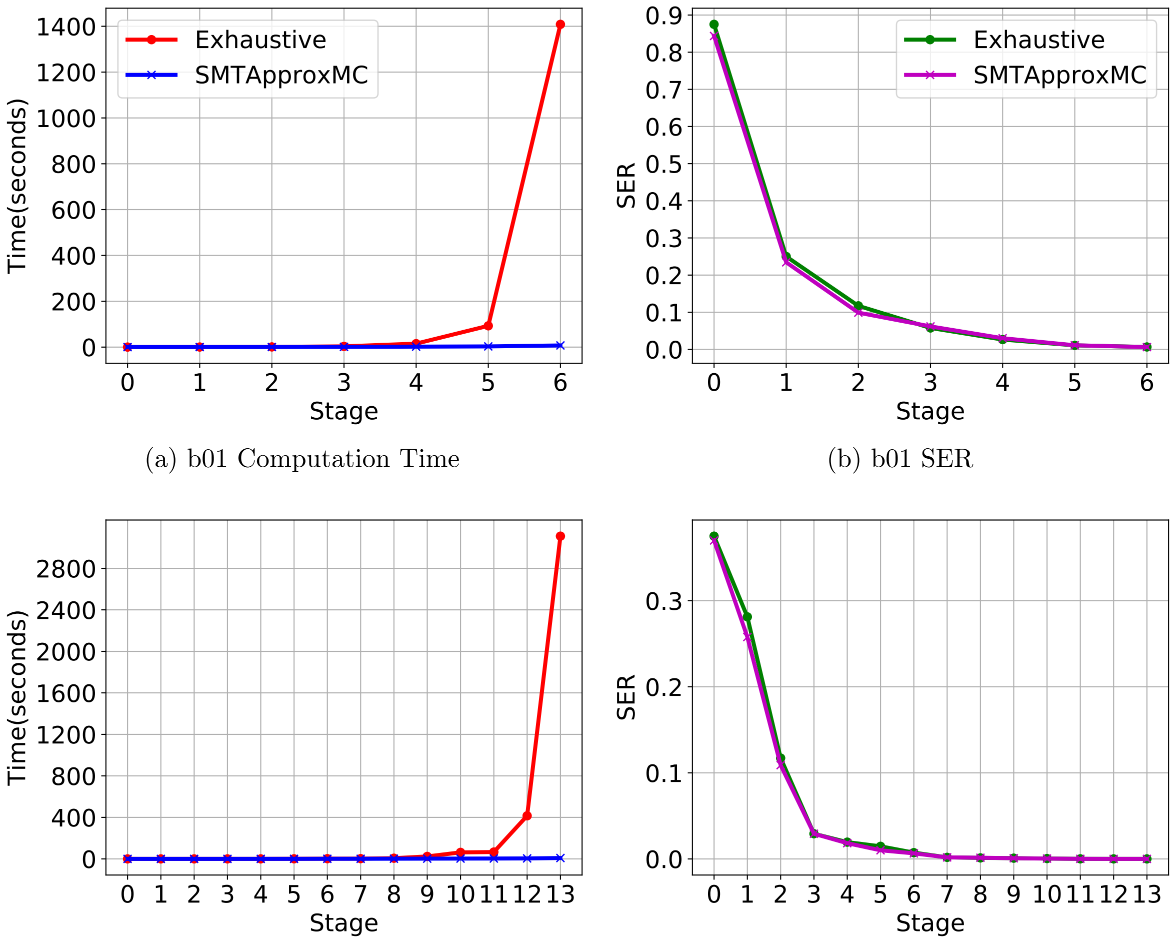
4.6 Experimental Results

The proposed analysis is performed using Python scripting and *Z3* [12] SMT solver. The SMTApproxMC tool [45] was used to generate approximate model counts. The proposed methodology was implemented on some ITC99 benchmark circuits [14]. Experiments were conducted on a workstation with an Intel(R) Core(TM) i7-6820HQ running at 2.70 GHz and with 16 GB RAM.

First, we investigate the computation time and accuracy of approximate model counting using the SMTApproxMC tool [45] by analyzing some circuits from the ITC99 benchmarks. In order to evaluate the accuracy, we computed the SERs using exhaustive model counting and approximate model counting. For the exhaustive method, we generate all satisfiable assignments using Z3's Python API. In order to do so, we generate a satisfiable assignment for the current injection scenario, and then add a new constraint that prevents the previous model from being generated again. This is repeated until the formula becomes unsatisfiable. For the benchmark circuits b01 (FSM that compares serial flows) and b02 (FSM that recognizes BCD numbers) the SER was computed for a certain number of stages following the SEU injection stage (stage 0). Figure 4.3 (a) and (c) show the computation time required to compute the SER using the exhaustive model counting versus approximate model counting using SMTApproxMC.

Using exhaustive model counting, the computation times are 1408 seconds and 3110 seconds for computing the SER of stage 7 of the b01 and stage 13 of the b02 respectively. On the other hand, computing the same SERs using SMTApproxMC [45] takes approximatively 7.2 seconds and 8.3 seconds respectively. Figure 4.3 (b) and (d) show the SER computed at every stage for the b01 and b02 using exhaustive model counting versus SMTApproxMC [45]. It can be observed that the loss in accuracy is on average 7.7% for the b01 and 0.1% for the b02. The SMTApproxMC [45] tool provides accurate results at a significant gain in computation time.

The analysis using SMTApproxMC [45] was also performed on other ITC99 benchmark circuits [14]. The results are presented in Table 7, which lists the tested circuits, the number of PIs and FFs of each circuit, the number of stages unrolled, the computer SER as well as the computation time. In our results, we unrolled an arbitrary number of times. The results demonstrate that the SER can be computed for several cycles following the SEU injection cycle. The computation time rises exponentially with the total search space which increases exponentially with the number of unrolled stages. The size of the search space for stage k is $2^{CS0+k*PI}$, where **CS0** represents



(a) b01 Computation Time

(b) b01 SER

(c) b02 Computation Time

(d) b02 SER

Figure 4.3: Comparison of the Computation Time and SER Between Exhaustive Model Counting and SMTApproxMC

the size of the current state registers of the initial stage (SEU injection stage) and **PI** the number of primary inputs. Small designs such as the b02 can be unrolled for 20 stages and require a computation time of 74 seconds while larger designs such as the b10 require 2432 seconds to unroll for 3 stages.

4.7 Summary

A new methodology based on SMT is proposed for estimating the soft-error vulnerability of sequential circuits at RTL. The proposed SMT model is able to capture fault

Table 7: Computation Times and SERs for ITC99 Benchmark Circuits

circuit	**PIs**	**POs**	**FFs**	**Stages**	**Comp. Time (s)**	**SER**
b01	2	2	5	11	354	5.28×10^{-4}
b02	1	1	4	20	74	3.92×10^{-7}
b03	4	4	30	3	1698	1.62×10^{-3}
b06	2	6	9	11	402	5.46×10^{-4}
b09	1	1	28	3	385	3.90×10^{-2}
b10	11	6	17	3	2432	9.77×10^{-11}

propagation properties directly at RTL descriptions of digital circuits. This allows to evaluate the effect of SEUs and compute the SER early in design stages. We have compared the trade-off in computation time versus accuracy of the latter and found out that a significant improvement in performance can be achieved (up to 37 times faster) at a reasonable cost in accuracy (as low as 0.1% error). Moreover, the technique was applied on some ITC99 benchmarks to demonstrate its applicability for investigating the vulnerability of sequential circuits early in the design stages.

Chapter 5

Efficient SMT Modeling and Analysis of SEUs Propagation in Combinational Circuits

5.1 Introduction

In this chapter, we introduce a new methodology to evaluate the vulnerability of digital circuits to soft errors due to SEUs at RTL. A new modeling of SEUs propagation as a Satisfiability problem using Satisfiability Modulo Theories (SMTs) is proposed. The behavior of the basic RTL operations (such as logical, reduction, arithmetic, and case statement) in the presence of SEUs is modeled.

The proposed methodology allows the analysis of SEUs propagation from each vulnerable node to the output. This is done by evaluating the generated SMT model of the RTL design against a set of assertions.

Using this efficient approach, our goal is to improve the scalability of the modeling proposed in Chapter 3. Moreover, we investigate the advantages of using SMT modeling over SAT modeling when dealing with SEU propagation in combinational circuits in more details. We take advantage of the improved scalability to analyze larger designs and compute a more detailed analysis of the propagation paths of SEUs. The computation times to analyze the ISCAS85 benchmarks [13] circuits at RTL are compared with the results provided in [1]. Moreover, to gain better insight on how much more efficient SMT modeling is over SAT modeling, we investigate the computation time required to analyze arithmetic circuits such as adders, subtractors, multipliers and dividers. Finally, we will compute the SERs based on our new approach with and without applying the same data type reduction technique presented in Section 3.2 to better understand how it affects the computer SER.

Our results demonstrate that our technique is more efficient than existing formal based techniques that use pure Boolean representation to model SEU propagation [1]. Moreover, the results demonstrate that our SMT modeling maintains a high level of accuracy compared to techniques such as [10], [2]. Experimental results demonstrate that the proposed framework is about 4 times faster than other comparable contemporary techniques. Moreover, it provides more accurate and detailed results of the circuit vulnerability allowing a more efficient applicability of fault tolerance techniques.

The rest of this chapter is organized as follows. Section 5.2 explains our new proposed modeling of SEUs propagation at RTL using only one copy of the circuit. In Section 5.3, we explain the analysis used in this chapter and the Soft Error Rate (SER) estimation approach. In Section 5.4 we use an exmaple to demonstrate the applicability of our proposed methodology. In Section 5.5, we explain our experiments and results. Section 5.6 concludes this work.

5.2 Modeling of RTL Constructs

In this section, we explain a new SMT modeling approach of SEU propagation in RTL combinational circuits. This approach will use only one copy of the design. The RTL signals and different constructs are modeled in a way that includes the fault propagation properties. In order to do so, a new library of SMT function is developed that keeps track of fault propagation while preserving the original functionality of the circuits.

The modeling of SEUs propagation through the basic RTL constructs is explained in details. In this section, we use capital symbols (e.g., A) to represent bit-vector variables. Reduction operators will be represented using single symbols (e.g., & for reduction AND and | for reduction OR).

5.2.1 Bit-Vector Signals

In order to keep track of the exact bits that are affected by an SEU, RTL bit-vectors are modeled using a data type consisting of two bit-vectors in SMT format. Say we have a 4-bit vector A, the first vector represents the logic state of the bits (e.g., A_l) and the second vector represents the fault state of the bits (e.g., A_f). For example, in Figure 5.1, A_l = "0100" and A_f = "0011" indicates that the error free logic state of bit-vector A is "0100" and there are two errors at bits 0 and 1, i.e., the faulty value of A is "0111".

5.2.2 Logical Operators

Logical operators are encoded as bit-vector operations over the logic and fault states of the input bit-vectors. The operations describing the propagation of faults include the case where the two inputs are faulty. This will never occur at the primary inputs, since only one SEU is injected at a time. This work does not consider MBUs (Multiple Bit Upsets). However, the case where the two inputs of an RTL Logical operator are faulty is needed to consider re-convergent faults. An SEU occurring at a primary input can re-converge into two errors on its propagation path to the outputs.

Logical AND Operator

For an AND operator with inputs A, B and output C, the logic state of C is given by:

$$C_l = A_l \wedge B_l$$

The bits of C_f are "1" (i.e., faulty) under any of the following two conditions. The first condition is if the corresponding bits of either input is faulty and the other is of non-controlling logic. For an AND operator, the non-controlling logic value is "1", i.e. if a bit is faulty and the other bit is of value "1", the fault propagates to the output of that specific bit. The second condition is if the corresponding bits of both inputs are faulty and of equal logic value. Therefore, the fault state of C is given by:

$$C_f = (A_f \wedge B_l \wedge \overline{B_f})|(B_f \wedge A_l \wedge \overline{A_f})|(A_f \wedge B_f) \wedge (\overline{A_l \oplus B_l})$$

In the example in Figure 5.1, C_l = "0100", which is the result of the logical AND operation over bit-vectors A_l and B_l. The fault state C_f = "0001" indicates that only the LSB of C_l is faulty, since the fault at $A_l[1]$ was logically masked due to the controlling logic value of B (i.e., $B_l[1]$ = "0").

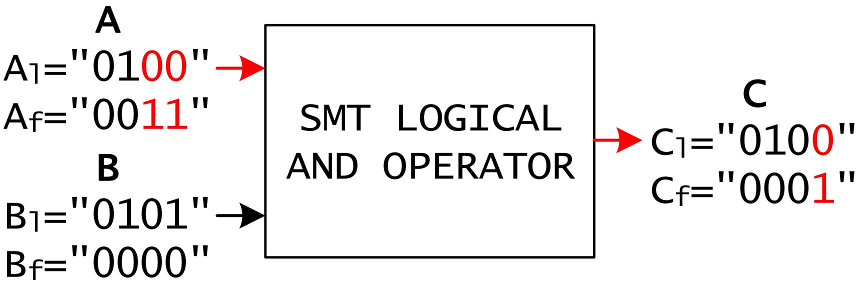


Figure 5.1: SMT Logical And Operator

Logical NAND Operator

For an $NAND$ operator with inputs A, B and output C, the logic state of C is given by:

$$C_l = \overline{A_l \wedge B_l}$$

Similar to the AND operator, the bits of C_f are "1" (i.e., faulty) under any of the following two conditions. The first condition is if the corresponding bits of either input is faulty and the other is of non-controlling logic. The second condition is if the corresponding bits of both inputs are faulty and of equal logic value. Therefore, the fault state of C for a $NAND$ operator is given by the same equation as that of a AND operator, i.e.:

$$C_f = (A_f \wedge B_l \wedge \overline{B_f})|(B_f \wedge A_l \wedge \overline{A_f})|(A_f \wedge B_f) \wedge (\overline{A_l \oplus B_l})$$

Logical OR Operator

For an OR operator with inputs A, B and output C, the logic state of C is given by:

$$C_l = A_l \vee B_l$$

The non-controlling logic for an OR operator is "0", i.e. if one bit is faulty, then the fault will only propagate to the output of the corresponding bit given the other bit is of value "0", otherwise it will be logically masked. Similarly to the AND and $NAND$ operators, the fault will also propagate given the two inputs are faulty and of equal value. Therefore, the fault state of C for a OR operator is given by:

$$C_f = (A_f \wedge \overline{B_l} \wedge \overline{B_f})|(B_f \wedge \overline{A_l} \wedge \overline{A_f})|(A_f \wedge B_f) \wedge (\overline{A_l \oplus B_l})$$

Logical NOR Operator

For an NOR operator with inputs A, B and output C, the logic state of C is given by:

$$C_l = \overline{A_l \vee B_l}$$

The non-controlling logic for an NOR operator is the same as for an OR operator, i.e. "0". Therefore, the fault state of C for a NOR operator is given by the same equation as that of an OR operator, i.e.:

$$C_f = (A_f \wedge \overline{B_l} \wedge \overline{B_f})|(B_f \wedge \overline{A_l} \wedge \overline{A_f})|(A_f \wedge B_f) \wedge (\overline{A_l \oplus B_l})$$

Logical XOR Operator

For an XOR operator with inputs A, B and output C, the logic state of C is given by:

$$C_l = A_l \oplus B_l$$

In an XOR operator, if only one of the inputs is faulty, the fault will always propagate. This can easily be seen from the truth table of an XOR operation. If the state of one bit is changed, the output will always change, regardless of the sate of the other bit. However, contrary to other operators, if both inputs happen to be faulty, the fault will be masked at the output. Therefore, the fault state of C for a XOR operator is given by:

$$C_f = (A_f \wedge \overline{B_f}) | (B_f \wedge \overline{A_f})$$

Which can be simplified as:

$$C_f = A_f \oplus B_f$$

Logical XNOR Operator

For an $XNOR$ operator with inputs A, B and output C, the logic state of C is given by:

$$C_l = \overline{A_l \oplus B_l}$$

Similarly to the XOR operator, a fault will only propagate to the output bits of an $XNOR$ operator given only one of the inputs is faulty. Therefore, the fault state of C for a XOR operator is given by:

$$C_f = A_f \oplus B_f$$

Logical NOT Operator

For a NOT operator with input A and output C, the logic state of C is given by:

$$C_l = \overline{A_l}$$

In a *NOT* operator, fault propagation is transparent, i.e., a faulty input always causes a faulty output. Therefore, the fault state of C for a *NOT* operator is given by:

$$C_f = A_f$$

Logical Operators SMT Function Table

The following table summarizes the SMT functions that represent the logic state and fault states of RTL logical operators.

Table 8: Logical Operators SMT Function Table

Logical	**Logic**	**Fault**
Operator	**State**	**State**
AND	$C_l = A_l \wedge B_l$	$C_f = (A_f \wedge B_l \wedge \overline{B_f})\vert(B_f \wedge A_l \wedge \overline{A_f})\vert(A_f \wedge B_f) \wedge (\overline{A_l \oplus B_l})$
NAND	$C_l = \overline{A_l \wedge B_l}$	$C_f = (A_f \wedge B_l \wedge \overline{B_f})\vert(B_f \wedge A_l \wedge \overline{A_f})\vert(A_f \wedge B_f) \wedge (\overline{A_l \oplus B_l})$
OR	$C_l = A_l \vee B_l$	$C_f = (A_f \wedge \overline{B_l} \wedge \overline{B_f})\vert(B_f \wedge \overline{A_l} \wedge \overline{A_f})\vert(A_f \wedge B_f) \wedge (\overline{A_l \oplus B_l})$
NOR	$C_l = \overline{A_l \vee B_l}$	$C_f = (A_f \wedge \overline{B_l} \wedge \overline{B_f})\vert(B_f \wedge \overline{A_l} \wedge \overline{A_f})\vert(A_f \wedge B_f) \wedge (\overline{A_l \oplus B_l})$
XOR	$C_l = A_l \oplus B_l$	$C_f = A_f \oplus B_f$
XNOR	$C_l = \overline{A_l \oplus B_l}$	$C_f = A_f \oplus B_f$
NOT	$C_l = \overline{A_l}$	$C_f = A_f$

5.2.3 Reduction Operators

These operators are unary i.e., they perform a bit-wise operation on one operand and produce a 1-bit output. In our model, the output of a reduction operator is faulty if at least one bit of the input vector is faulty, all the non-faulty bits are of non-controlling logic and the logic states of all the faulty bits are equal.

Reduction AND Operator

For example, for a reduction AND with input A and output B, the logic state of B is the reduced AND operation over the logic state of A as follows:

$$B_l = \&A_l$$

Before showing the fault state of output B for a reduction AND operator, we will use the following example. If we have a 4-bit input A with the following logic and fault states:

$$A_l = \texttt{"0101"}$$

$$A_f = \texttt{"1010"}$$

For B to be faulty ($B_f = \texttt{"1"}$) the following two conditions must be satisfied. First, at least one bit of A is faulty which is determined by performing a reduced OR operation over A_f. Second, all the non-faulty bits have a "1" logic value and the logic state of all the faulty bits is equal. This second conditions has two cases:

1. The first case is that the non-faulty bits are "1" and faulty bits are "0". The formula representing this condition is $\&(A_l \oplus A_f)$.

2. The second case is that the non-faulty bits are "1" and faulty bits are "1". The formula representing this condition is $\&A_l$.

Therefore, the fault state of B is given by:

$$(|A_f) \wedge (\&(A_l \oplus A_f) \vee \&A_l)$$

The correct output of the reduction AND operation on the bit-vector A_l is "0". However, since bits 3 and 1 are faulty, using the equation describing the fault state of B, the output of B is found to be faulty, i.e. $B_f = \texttt{"1"}$. This is because, a bit flip at bits 3 and 1, will result in the logic state of A being "1111". Therefore the reduced AND operation will output a value of "1" instead of "0".

For the above example, the output B will be given by the following logic and fault states:

$$B_l = "0"$$

$$B_f = "1"$$

Reduction OR Operator

For example, for a reduction OR with input A and output B, the logic state of B is the reduced OR operation over the logic state of A as follows:

$$B_l = |A_l$$

If we have a 4-bit input A with the following logic and fault states:

$$A_l = "0000"$$

$$A_f = "0100"$$

For B to be faulty ($B_f = "1"$) the following two conditions must be satisfied. First, at least one bit of A is faulty which is determined by performing a reduced OR operation over A_f. Second, all the non-faulty bits have a "0" logic value and the logic state of all the faulty bits is equal. This second conditions has two cases:

1. The first case is that the non-faulty bits are "1" and faulty bits are "0". The formula representing this condition is $\&(\overline{A_l} \oplus A_f)$.
2. The second case is that the non-faulty bits are "1" and faulty bits are "1". The formula representing this condition is $\&\overline{A_l}$.

Therefore, the fault state of B is given by:

$$(|A_f) \wedge (\&(\overline{A_l} \oplus A_f) \vee \&\overline{A_l})$$

The correct output of the reduction OR operation on the bit-vector A_l is "0". However, since bit 2 is faulty, using the equation describing the fault state of B, the output of B is found to be faulty, i.e. $B_f =$ "1". This is because, a bit flip at bit 2, will result in the logic state of A being "0100". Therefore the reduced OR operation will output a value of "1" instead of "0".

For the above example, the output B will be given by the following logic and fault states:

$$B_l = \text{"0"}$$

$$B_f = \text{"1"}$$

5.2.4 Arithmetic Operators

The SMT theory of bit-vectors allows arithmetic operations to be performed on bit-vectors. It is therefore possible to perform the multiplication, division, addition, and subtraction operations (i.e. *, /, +, -) on both the faulty and non-faulty values of the signals to keep track of the faulty bits. To obtain the faulty value of signal A, we simply perform a logical XOR operation with its fault state:

$$A_l \oplus A_f$$

In the case of an addition $C = A + B$, the non-faulty result is evaluated using as:

$$C_l = A_l + B_l$$

The faulty result is evaluated using $(A_l \oplus A_f) + (B_l \oplus B_f)$. From this, the fault state of C is computed by performing a logical XOR operation over the two vectors:

$$C_f = (A_l + B_l) \oplus ((A_l \oplus A_f) + (B_l \oplus B_f))$$

5.2.5 If Statements and Case Statements

These are modeled using Boolean implication operators (i.e., =>) or using if-then-else (i.e., *ite*) operators. In order to keep track of the faulty bits in the output, if the select signal is faulty, both the faulty and non-faulty outputs are compared to compute the fault state vector.

5.2.6 Concatenation, Extraction, and Extension

For concatenation operations, i.e., $A :: B$, the logic states and the fault states are concatenated using the SMT concatenation operator as follows:

$$A_l :: B_l$$

$$A_f :: B_f$$

For bit extraction, the bits are simply extracted along with their corresponding fault bits using the SMT extract operator. For signed and unsigned extension, an SMT sign-extend and unsigned-extend operation is applied on both the logic and fault states of the bit-vectors respectively. For example, given:

$$A_l = \texttt{"1011"}$$

$$A_f = \texttt{"1000"}$$

If we apply a 4-bit sign extension, the result is:

$$A_l = \texttt{"11111011"}$$

$$A_f = \texttt{"11111000"}$$

5.3 Analysis of SEUs Propagation and SER Estimation

The main steps of our methodology are shown in Figure 5.2. We start with an RTL description of the circuit we wish to analyze, then we apply the data type reduction technique if desired. The SMT Model in Python is then generated using the SMT Library of basic RTL functions that was described in the previous section. Following that, we inject an SEU and compute the propagation probabilities either using an exhaustive model counting approach or the SMTApproxMC tool [45]. These steps are repeated for all injections scenarios.

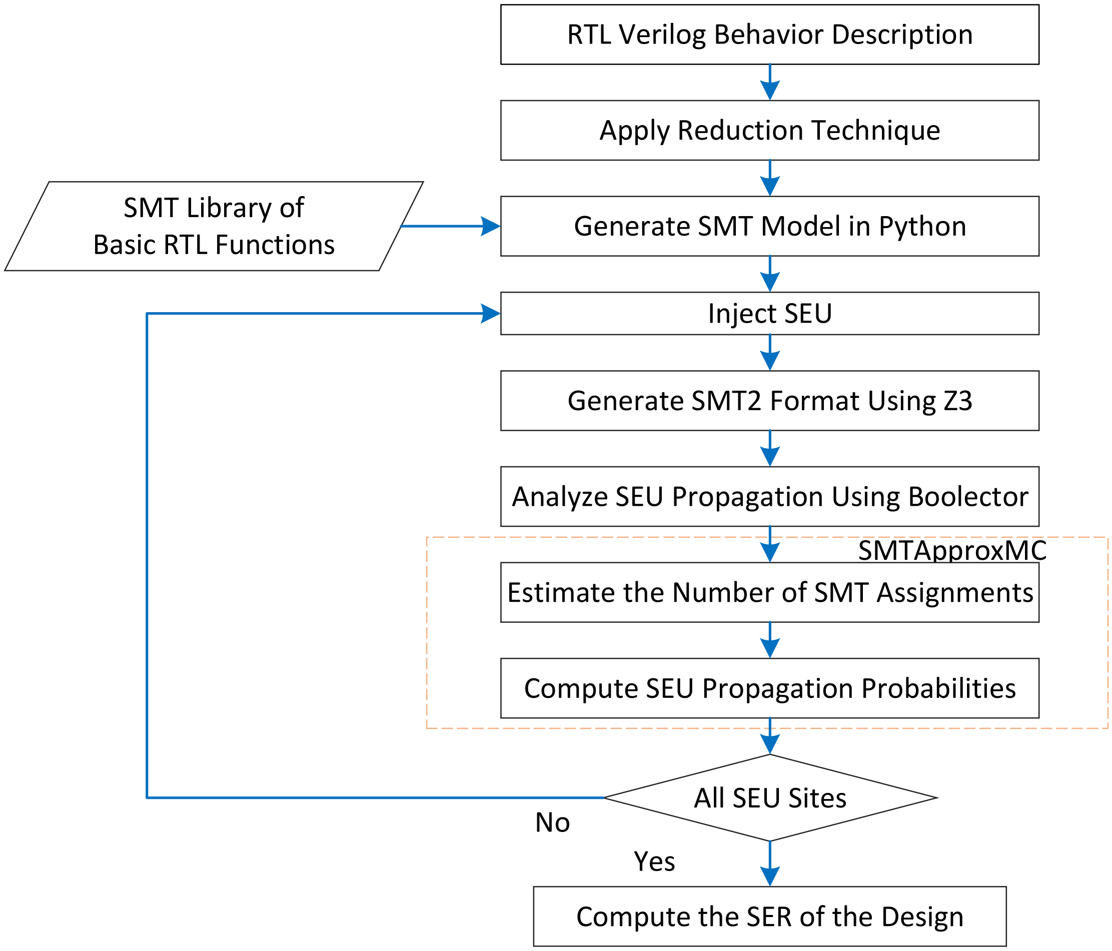

RTL Verilog Behavior Description
Apply Reduction Technique
SMT Library of Basic RTL Functions
Generate SMT Model in Python
Inject SEU
Generate SMT2 Format Using Z3
Analyze SEU Propagation Using Boolector
SMTApproxMC
Estimate the Number of SMT Assignments
Compute SEU Propagation Probabilities
All SEU Sites
No
Yes
Compute the SER of the Design

Figure 5.2: Main Steps of the Proposed Modeling and Analysis of SEU Propagation at RTL

The first step in our methodology, is to convert the RTL behavioral Verilog description into a Python SMT model using our basic RTL function libraries. Those libraries were developed using the Microsoft Z3 [12] Python API. Then, SEU is injected at one bit by setting its corresponding fault state bit to "1". The next step is to convert the design into the SMT2 format, which is then fed to the SMTApproxMC tool [45] to perform the following tasks: 1) verify SEUs propagation to the output using the Boolector SMT solver; 2) estimate the number of input vectors that allow the SEU injected at node i to reach output j, which is equal to the number of satisfiability assignments (i.e., $Num\_Assignments_{i \longrightarrow j}$). Thereafter, the probability that an injected SEU at input i propagates to an output j is computed as shown in equation (6).

$$P(SEU_{i\longrightarrow j}) = \frac{Num\_Assignments_{i\longrightarrow j}}{N} \tag{6}$$

In equation (6), $Num\_Assignments_{i\longrightarrow j}$ is the number of satisfiable SMT assignments for the current injection scenario and N is the total number of input vectors. This analysis is performed for all possible inputs and outputs combinations. Therefore, the vulnerability of an output j for a circuit which has m vulnerable nodes, is calculated as shown in equation (7).

$$Vuln(j) = \sum_{i=0}^{m-1} P(SEU_{i\longrightarrow j}) \tag{7}$$

Finally, the Soft Error Rate (SER) of a circuit, which has n outputs, is calculated as shown in equation (8).

$$SER = \sum_{j=0}^{n-1} Vuln(j) \tag{8}$$

5.4 Example

As an example, the proposed methodology was implemented on the 74L85 4-bit magnitude comparator circuit shown in Figure 5.3 to illustrate its main steps and investigate the results. This circuit implements a magnitude comparator by a carry function with an inverted input. In this circuit, common elements of the three comparator functions, i.e., $A < B$, $A > B$ and $A = B$ are combined.

A high level description of the circuit in Verilog was converted into an SMT equivalent formula using the developed libraries. The propagation probabilities from every input to every output were evaluated. These probabilities are shown in Table 9. Using the probabilities obtained from this analysis, it is then possible to identify the most critical bits (i.e, input and output flip-flops) for SEU propagation. In other words, we can identify the inputs from which SEUs have the highest probabilities

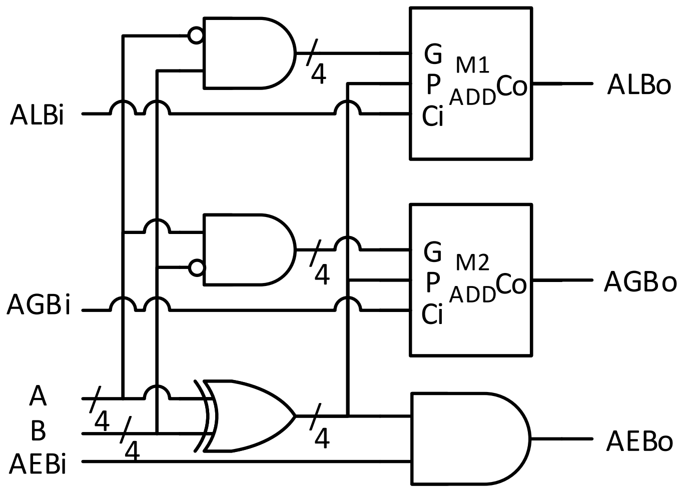


Figure 5.3: RTL Structure of the 74L85 Benchmark that is a 4-bit Magnitude Comparator

to propagate to a given output. Using these results, it is also possible to identify not only the most vulnerable bits, but the propagation paths as well. For example, SEUs injection at inputs $A[3]$ and $B[3]$ have the highest probability to propagate to $ALBo$ and $AGBo$. Such results can be very useful for any SEU tolerant technique to selectively harden the most vulnerable bits in order to achieve the desired SER with minimum area overhead.

The probability of SEU propagation from each input bit to each output is computed based on equation (6). Using equation (7), the vulnerability of each output is computed and the results are reported in the last row of Table 9. Thereafter, the SER of the circuit is evaluated to be equal to 4.4375 based on equation (8). Note that vulnerabilities are sums of probabilities that are not disjoint and they can add up to values larger than 1.

In Chapter 3, an SMT implementation of the two-versions technique was proposed. The goal of the new modeling approach proposed in the current chapter, is to

Table 9: SEU Propagation Probabilities and Derived Vulnerabilities for the 74L85 4-bit Magnitude Comparator Circuit

	ALBo	**AGBo**	**AEBo**	**Vulnerability**
A[3]	0.5	0.5	0.0625	1.0625
A[2]	0.25	0.25	0.0625	0.5625
A[1]	0.125	0.125	0.0625	0.3125
A[0]	0.0625	0.0625	0.0625	0.1875
B[3]	0.5	0.5	0.0625	1.0625
B[2]	0.25	0.25	0.0625	0.5625
B[1]	0.125	0.125	0.0625	0.3125
B[0]	0.0625	0.0625	0.0625	0.1875
ALBi	0.0625	0	0	0.0625
AGBi	0	0.0625	0	0.0625
AEBi	0	0	0.0625	0.0625
Vulnerability	1.9375	1.9375	0.5625	4.4375

investigate how using only one copy of the design by including the fault propagation properties within the model can improve the analysis.

The statistics class of Microsoft Z3 SMT solver allows to track statistical information about the solver objects. For example, we can track the amount of memory used by a solver object to solve a given set of assertions. It is also possible to track the number of conflicts encountered by the solver to find the satisfiable assignments. The number of conflicts indicate the number of assignments made by the theory subsolvers that make the formula false. When checking a formula for satisfiability, the solver object attempts to find a satisfiable assignment. Given a list of clauses to satisfy, the solver picks a variable, sets it to a value, and repeats this for all variables while building a decision graph. If setting a variable to a certain value leads to a conflict, i.e. certain clauses become unsatisfiable, the solver backtracks to the previous level in the decision graph, and sets the variable to another value. If the formula is satisfiable, a high number of conflicts means that the solver tried a lot of assignments for different variables that did not satisfy the formula, which means that the solver did not explore the search space efficiently. A high number of conflicts implies a larger portion of the search space was traversed by Z3.

In order to investigate the difference between two-versions modeling proposed in chapter 3 and the modeling proposed in the current chapter, we analyzed the computation time, memory consumption and the number of conflicts that the solver encountered when exhaustively computing all satisfiable assignment for the 74L85. When using the technique proposed in this chapter, our results demonstrate a speed-up in computation time of 1.1. Moreover, we observe a decrease in the number of conflicts of 82% and a decrease in memory consumption of about 2%.

In the next subsection we analyze more ISCAS85 benchmarks using the tool SMTApproxMC [45]. We will compare our results to the results in [1]. We will observe that when using approximate model counting and analyzing much larger circuit, the improvement in computation time becomes much larger.

5.5 Experimental Results

In this section, we report results from experiments performed to validate the proposed methodology and its efficiency. The proposed modeling and analysis is fully automated using the Z3 SMT solver [12], the SMTApproxMC tool [45], and Python scripts. Our experiments were performed on a workstation with an Intel(R) Core(TM) i7-6820HQ running at 2.70 GHz and with 16 GB RAM.

Typically, SEU analysis is done using two versions of a circuit, a fault-free version and a faulty version. A fault is injected by flipping a single bit in the faulty version of the circuit. The outputs of the two versions are then compared to check if the SEU reached the output. The accuracy of the two-versions modeling approach was proven in [1]. In order to validate the accuracy of our technique, we implemented the two-versions based modeling using SMT to compare the results with the modeling proposed in Section 5.2. Moreover, we applied the data type reduction described in Section 3.2 on our model in order to evaluate the loss in accuracy due to this reduction technique. It is assumed that the inputs of the circuits are latched, i.e., the data comes from registers, where the SEUs will be injected.

5.5.1 Performance Analysis

The first detailed analysis consisted of investigating the applicability and the performance of our methodology. Different behavioral descriptions of the ISCAS85 benchmarks [13] and MSI components in the 74xxx series were analyzed. The results are reported in Table 10. The computation times are compared with the results presented in [1] (shown in the second column) which are based on pure Boolean SAT. Moreover, we compare the computation time of the modeling proposed in Section 5.2 with the computation time of the two-versions modeling that was implemented using SMT as well. As expected, it can be observed that the proposed SMT modeling and analysis provides a better solution to analyze SEUs at RTL in comparison with the pure Boolean SAT implementation. It is observed that our methodology is on average about 4 times faster.

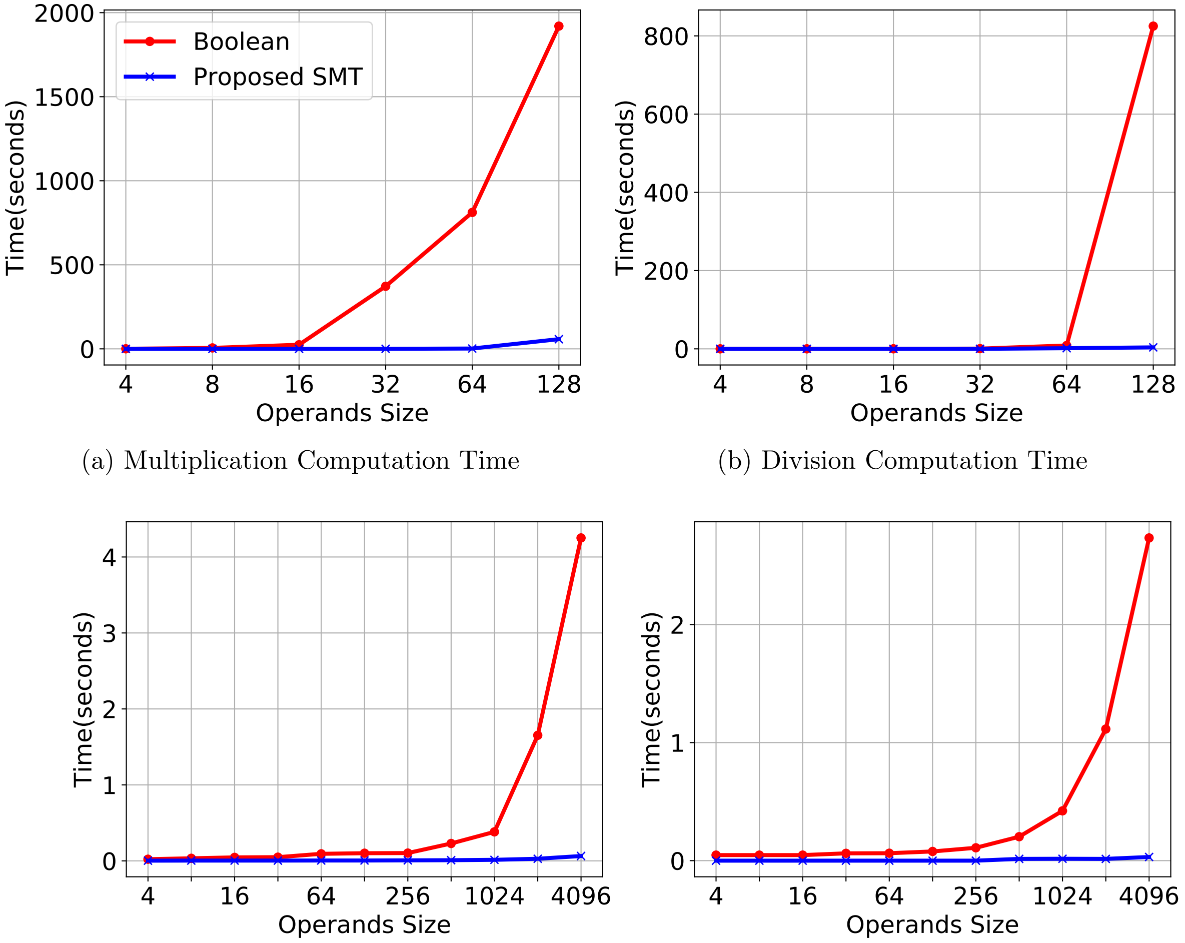
On the other hand, analyzing the fault propagation of SEUs using a fault-free and faulty versions of the RTL circuit implicitly doubles the number of inputs and operations in the resulting model. Therefore, this technique results in significant modeling redundancy. As discussed in the example of Section 5.4, using the approach proposed in this chapter over using two versions of the designs reduces the size of the formula and reduces the number of conflicts encountered by the SMT solver. Moreover, using our proposed modeling instead of two-versions modeling significantly

Table 10: Comparison of the Computation Time between the Proposed Methodology, the Technique Proposed in [1] and the Two-Versions Technique

Circuit	**Technique in [1] Comp. Time (s)**	**Two-Versions Comp. Time (s)**	**Proposed Methodology Comp. Time (s)**
74283	0.27	0.25	0.13
74182	0.22	0.15	0.11
74181	9.36	7.01	1.29
74L85	8.68	6.45	1.22
c432	266.80	245.22	114.22
c499	46.80	42.11	22.92
c1908	-	432.64	345.78
c2670	-	733.43	548.54
c3540	-	335.11	245.98

reduced the computation time.

Another analysis was conducted to compare our technique with the technique proposed in [1] for larger designs. In this analysis, we implemented the modeling proposed in [1] using the SAT formulation and our proposed modeling using the SMT formulation. Results demonstrate that with SAT based modeling, the computation time grows exponentially with the design size. Figure 5.4 shows the time required to analyze the propagation of an SEU injected at the input of arithmetic circuits using the modeling proposed in [1] (called Boolean in Figure 5.4(a)) and our proposed modeling for various circuit sizes. For example, analyzing SEUs propagation for a 128-bit multiplication using the technique in [1] can take up to 1920.0 seconds, whereas



(a) Multiplication Computation Time

(b) Division Computation Time

(c) Addition Computation Time

(d) Subtraction Computation Time

Figure 5.4: Comparison of the Computation Time Between the Boolean Based Modeling and the Proposed SMT Modeling for Different Sizes of Arithmetic Operations

analyzing the same size operation using our technique takes only 57.6 seconds.

5.5.2 Accuracy Analysis

The accuracy of our method was compared with the technique proposed in [2], which employs a data reduction technique. This comparison was done in Section 3.4. However, in the current section, we use a different formula to compute the SER. For this reason, we did another comparison to investigate the impact of using our adaptation of the data type reduction technique that was discussed in Section 3.2.

We also validate our results by comparing the SERs computed with our methodology with those computed with the two-versions technique. In Table 11, the results in the first column represent the SER values computed when using the data type reduction technique [2] discussed in Section 3.2. The two-versions modeling and the modeling proposed in Section 5.2 do not use any reduction. Therefore, comparing the results of column 1 to those of columns 2 and 3, we can observe a large inaccuracy in the computed SER when using the data reduction technique.

Table 11: Comparison of the Computed Soft Error Rate (SER) Between the Proposed Methodology, the Technique Proposed in [2] and the Two-Versions SMT Modeling

circuit	**Technique in [2]** **SER**	**Two-Versions** **SER**	**Proposed** **Methodology** **SER**
74283	22.5	16.1	16.1
74182	20.75	5.72	5.72
74181	28.0	18.31	18.31
74L85	13.5	4.44	4.44
c432	25.56	13.89	13.50
c499	32.01	31.23	32.22
c1908	95.65	78.88	81.53
c2670	157.65	127.54	130.00
c3540	96.81	41.59	44.70

We illustrate the reason behind the loss in accuracy when using the data type reduction technique by analyzing the Program Counter (PC) address datapath of the MIPS architecture shown in Figure 5.5. The main issue when using data type reduction is that it assumes all the bits in an input register have equal propagation probabilities. Based on the technique proposed in [2], it is estimated that all SEUs injected at PC propagate through this datapath with a probability of 1. Our analysis shows that these results provide an over approximation of the actual SEU propagation probabilities in this circuit. Given equal probability of R-type, jump, and branch instructions, our results show that an SEU at the 2 LSBs will only propagate given R-type or branch instructions (i.e., probability of 2/3). An error occurring at bits [27 : 2] has a propagation probability slightly above 2/3 since these bits are always used for regular and branch type instructions. SEUs that occur when the current instruction is a jump instruction have a small propagation probability through the addition. In Pseudo-Direct Addressing for a jump instruction, the current PC is incremented by 4, then the four most significant bits are concatenated with the 26 least significant bits of the jump instruction, shifted by 2 to the left i.e., $PC \leftarrow PC[31:28] :: IR[25:0] :: 00$. In this case, an SEU occurring at bits [27 : 2] of the current PC address will be masked if the current instruction is a jump and the error does not propagate through the carries to reach the four MSBs. An SEU occurring on the four MSBs of the PC address, will always propagate to the computed effective address, because these 4 bits are always used to compute the next PC address. This analysis illustrates the accuracy of our technique that provides detailed vulnerabilities of the specific register bits.

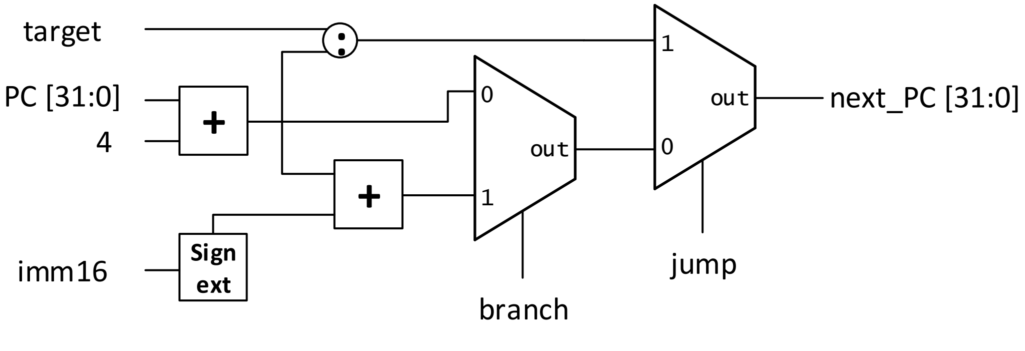


Figure 5.5: Program Counter Datapath

Columns 2 and 3 of table 11 represent the SERs computed using the two-versions technique and the modeling proposed in Section 5.2. The small difference in computed SER values can be justified by the model count estimation technique used in the SMTApproxMC tool [45], and not by the modeling accuracy. For small circuits where the SMTApproxMC tool [45] is able to generate all solutions, then the SER generated using the proposed methodology is exactly equal to the SERs of the two-versions models. However, for large circuits, the SMTApproxMC tool [45] randomly partitions the solution space of a given formula into smaller cells of roughly the same size using word-level hash functions. This causes small variations in the computed probabilities of two equivalent formulas.

5.6 Summary

In this chapter, we proposed a new methodology to investigate the vulnerability of combinational circuits to SEUs at RTL. SMT libraries of RTL constructs (e.g., arithmetic operations and conditional statements) are built to model SEU propagation. The proposed modeling significantly reduces the complexity of the analysis compared to other techniques while improving the accuracy. For instance, the CPU time required to compute the SER of ISCAS85 RTL benchmarks is on average about 4 times faster than with other techniques that use pure Boolean SAT to model SEU propagation. Moreover, this speed-up increases exponentially with larger circuits such as multipliers. The proposed technique improves the accuracy by providing detailed vulnerability information for specific bits in comparison with techniques that apply data type reduction.

Chapter 6

Conclusion and Future Work

In this work, we presented a methodology to analyze SEU propagation in digital circuits at RTL using SMTs. The purpose of our work was to investigate the efficiency of SMT modeling for the analysis of SEU propagation. In Chapter 3 we presented a first approach to analyze the vulnerability of combinational circuits to SEUs using an SMT modeling that uses two versions of the design. In Chapter 4 we extended our modeling technique to be applicable to sequential circuits. In Chapter 5, we enhanced the modeling technique presented in Chapter 3 and presented a new technique that uses only one version of the design.

6.1 Conclusion

In Chapter 3, we presented our first modeling approach to investigate SEU propagation in combinational circuits. We converted some combinational circuits from the ISCAS85 benchmark into SMT format. We then used to copies of the design, one fault-free and one faulty version where we injected an SEU. This technique is widely used in the current literature for fault simulation. In this first approach, we computed the vulnerability of the circuits by exhaustively generating all satisfiable assignments to a given SMT formula that represents the circuits under test. This approach proved to be inefficient due to exponentially increasing input vector search space. To overcome this, a randomly restricted input search was used at a cost in accuracy. Moreover, we investigated the effect of using a data type reduction technique on our modeling. We adapted our own version of the data type reduction technique presented in [2] into our SMT model and compared the results. Our comparison showed that this reduction technique resulted in some performance improvement but at a significant cost in accuracy.

In Chapter 4, the modeling approach proposed in Chapter 3 was extended to analyze sequential circuits. This approach still uses two versions of the design to investigate the propagation of SEUs. To analyze SEU propagation across multiple cycles, we adapted the circuit unrolling on our SMT model. Moreover, in this chapter we investigated the efficiency of approximate model counting over the exhaustive method proposed in Chapter 3. In order to do so, we used the recently used SMT approximate model counting tool SMTApproxMC [45]. This approach proved to be efficient as it significantly improved the computation time required to analyze some sequential RTL circuits from the ITC99 benchmark. This performance improvement was achieved with minimal impact on the accuracy of the computed SER.

In Chapter 5, we presented a more efficient modeling approach than the technique presented in Chapter 3. In this approach, instead of using two copies of the design and compare the outputs, only one copy is used. The fault propagation properties are embedded within the signals themselves. A new library mapping the RTL operations to SMT operations is defined. The new defined functions take as input the logic state as well as the fault state of signals. This modeling approach proved to be an improvement over the first approach used. It resulted in better computation times. Moreover, some of the SMT Solver characteristics were enhanced as well, such as memory consumption and number of conflicts encountered.

6.2 Future Work

Some of the worth mentioning extensions of our work are outlined as follows:

- The purpose of gaining information regarding the vulnerability of digital circuits early in the design stages is to apply faulty mitigation techniques. Such techniques include Triple Modular Redundancy (TMR) [9] or gate hardening. Our technique allows an efficient analysis of the most vulnerable path for SEU propagation at RTL. Using this information, it is possible to perform a more targeted application of TMR on the most vulnerable paths. As a future work, our methodology can be extended to identify those paths and apply TMR to obtain the best trade-off in terms of SER reduction and area overhead.

- The data type reduction approach in this work is a naive approach to reducing the circuit under test for SEU propagation analysis. One possible enhancement would be to enhance this reduction technique to minimize or eliminate the accuracy lost in the analysis while maintaining significant performance improvement. The improved data type reduction should allow to reduce the redundant parts of a circuit to improve performance, while still providing the ability to compute the vulnerability of every individual register bit. One possible idea, is to identify the bits which have a common operation on their path and reduce those specific bits into one bit, rather than reducing whole signals into one bits.

- Another possible area of improvement is regarding SMT model counting. Although not related to our modeling approach or to SEU propagation analysis specifically, SAT and SMT model counting is a well known area of interest. Our approach can handle fairly large circuits, but if we want to deal with very large designs, it is imperative to dig in the area of model counting and improve the current techniques available.

- Our proposed technique currently handles SEUs only. Nevertheless it can easily be extended to analyze Multiple Bit Upsets (MBUs). It is only required to modify the initial fault injection assertions to do so. Instead of asserting single bits to be faulty, the tool can be modified to assert multiple bits to be faulty and investigate their effect on combinational and sequential circuits.

Bibliography

[1] Syed Zafar Shazli and Mehdi Baradaran Tahoori. Using boolean satisfiability for computing soft error rates in early design stages. *Microelectronics Reliability*, 50(1):149–159, 2010.

[2] Liang Chen, Mojtaba Ebrahimi, and Mehdi B Tahoori. Formal quantification of the register vulnerabilities to soft error in RTL control paths. *Journal of Electronic Testing*, 31(2):193–206, 2015.

[3] F Sturesson. Single event effects (SEE) mechanism and effects. *Space Radiation and its Effects on EEE Components*, 2009.

[4] Rémi Gaillard. Single event effects: Mechanisms and classification. In *Soft Errors in Modern Electronic Systems*, pages 27–54. Springer, 2011.

[5] Tanay Karnik and Peter Hazucha. Characterization of soft errors caused by single event upsets in cmos processes. *IEEE Transactions on Dependable and Secure Computing*, 1(2):128–143, 2004.

[6] Huichu Liu, Matthew Cotter, Suman Datta, and Vijaykrishnan Narayanan. Soft-error performance evaluation on emerging low power devices. *IEEE Transactions on Device and Materials Reliability*, 14(2):732–741, 2014.

[7] Tomas Zaremba, Annette Ross Jakobsen, Mette Søgaard, Anna Margrethe Thøgersen, and Sam Riahi. Radiotherapy in patients with pacemakers and implantable cardioverter defibrillators: a literature review. *Europace*, 18(4):479–491, 2016.

[8] PD Bradley and E Normand. Single event upsets in implantable cardioverter defibrillators. *IEEE Transactions on Nuclear Science*, 45(6):2929–2940, 1998.

[9] Robert E Lyons and Wouter Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

[10] Jayanand Asok Kumar. *Statistical guarantees of performance for RTL designs*. PhD thesis, University Of Illinois At Urbana-Champaign, 2012.

[11] PRISM. PRISM - Probabilistic Symbolic Model Checker. `http://www.prismmodelchecker.org`, 2017.

[12] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[13] Mark C Hansen, Hakan Yalcin, and John P Hayes. Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering. *IEEE Design & Test of Computers*, 16(3):72–80, 1999.

[14] Fulvio Corno, Matteo Sonza Reorda, and Giovanni Squillero. RT-level ITC'99 benchmarks and first atpg results. *IEEE Design & Test of computers*, 17(3):44–53, 2000.

[15] Ghaith Kazma, Ghaith Bany Hamad, Otmane Ait Mohamed, and Yvon Savaria. Investigating the efficiency and accuracy of a data type reduction technique for soft error analysis. In *Electronics, Circuits and Systems (ICECS), 2016 IEEE International Conference on*, pages 273–276. IEEE, 2016.

[16] Ghaith Kazma, Ghaith Bany Hamad, Otmane Ait Mohamed, and Yvon Savaria. Analysis of SEU propagation in combinational circuits at RTL based on satisfiability modulo theories. In *Great Lakes Symposium on VLSI (GLSVLSI), 2017 International*. IEEE, 2017.

[17] Ghaith Kazma, Ghaith Bany Hamad, Otmane Ait Mohamed, and Yvon Savaria. Analysis of SEU propagation in sequential circuits at RTL using satisfiability modulo theories. In *New Circuits and Systems Conference (NEWCAS), 2017 15th IEEE International*. IEEE, 2017.

[18] Juan-Carlos Baraza, Joaquín Gracia, Sara Blanc, Daniel Gil, and Pedro-J Gil. Enhancement of fault injection techniques based on the modification of vhdl code. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(6):693–706, 2008.

[19] Kanad Basu, Prabhat Mishra, and Priyadarsan Patra. Observability-aware directed test generation for soft errors and crosstalk faults. In *VLSI Design and 2013 12th International Conference on Embedded Systems (VLSID), 2013 26th International Conference on*, pages 291–296. IEEE, 2013.

[20] Smita Krishnaswamy, Igor L Markov, and John P Hayes. Improving testability and soft-error resilience through retiming. In *Proceedings of the 46th Annual Design Automation Conference*, pages 508–513. ACM, 2009.

[21] Smita Krishnaswamy, Stephen M Plaza, Igor L Markov, and John P Hayes. Enhancing design robustness with reliability-aware resynthesis and logic simulation. In *Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 149–154. IEEE Press, 2007.

[22] Ghaith Bany Hamad, Syed Rafay Hasan, Otmane Ait Mohamed, and Yvon Savaria. Characterizing, modeling, and analyzing soft error propagation in asynchronous and synchronous digital circuits. *Microelectronics Reliability*, 55(1):238–250, 2015.

[23] Natasa Miskov-Zivanov and Diana Marculescu. Mars-c: modeling and reduction of soft errors in combinational circuits. In *Proceedings of the 43rd annual Design Automation Conference*, pages 767–772. ACM, 2006.

[24] Liang Chen, Mojtaba Ebrahimi, and Mehdi B Tahoori. Cep: correlated error propagation for hierarchical soft error analysis. *Journal of Electronic Testing*, 29(2):143–158, 2013.

[25] Ghaith Bany Hamad, Ghaith Kazma, Otmane Ait Mohamed, and Yvon Savaria. Efficient and accurate analysis of single event transients propagation using smt-based techniques. In *Proceedings of the 35th International Conference on Computer-Aided Design*, page 54. ACM, 2016.

[26] Xiaodong Li, Sarita V Adve, Pradip Bose, and Jude A Rivers. Softarch: an architecture-level tool for modeling and analyzing soft errors. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 496–505. IEEE, 2005.

[27] Cadence. Cadence SMV Model Checker. `http://www.kenmcmil.com/smv.html`, 2017.

[28] AIGER. Aiger libraries. `http://fmv.jku.at/aiger/`, 2017.

[29] Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In *International Conference on Computer Aided Verification*, pages 17–36. Springer, 2002.

[30] João P Marques-Silva and Karem A Sakallah. Boolean satisfiability in electronic design automation. In *Proceedings of the 37th Annual Design Automation Conference*, pages 675–680. ACM, 2000.

[31] Niklas Sorensson and Niklas Een. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005(53), 2005.

[32] Niklas Sörensson and Niklas Eén. Minisat 2.1 and minisat++ 1.0-sat race 2008 editions. *SAT*, page 31, 2009.

[33] Armin Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.

[34] Knot Pipatsrisawat and Adnan Darwiche. Rsat 2.0: Sat solver description. *SAT competition*, 7, 2007.

[35] Gilles Audemard and Laurent Simon. Glucose: a solver that predicts learnt clauses quality. *SAT Competition*, pages 7–8, 2009.

[36] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185:825–885, 2009.

[37] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer, 2009.

[38] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2):1–2, 2006.

[39] Bruno Dutertre. Yices 2.2. In *International Conference on Computer Aided Verification*, pages 737–744. Springer, 2014.

[40] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[41] Wei Wei and Bart Selman. A new approach to model counting. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 324–339. Springer, 2005.

[42] Stefano Ermon, Carla P Gomes, and Bart Selman. Uniform solution sampling using a constraint solver as an oracle. *arXiv preprint arXiv:1210.4861*, 2012.

[43] Reuven Rubinstein. Stochastic enumeration method for counting np-hard problems. *Methodology and Computing in Applied Probability*, 15(2):249–291, 2013.

[44] Vibhav Gogate and Rina Dechter. Samplesearch: Importance sampling in presence of determinism. *Artificial Intelligence*, 175(2):694–729, 2011.

[45] Supratik Chakraborty, Kuldeep S Meel, Rakesh Mistry, and Moshe Y Vardi. Approximate probabilistic inference via word-level counting. *arXiv preprint arXiv:1511.07663*, 2015.

[46] Natasa Miskov-Zivanov and Diana Marculescu. Mars-s: Modeling and reduction of soft errors in sequential circuits. In *Quality Electronic Design, 2007. ISQED'07. 8th International Symposium on*, pages 893–898. IEEE, 2007.

[47] Natasa Miskov-Zivanov and Diana Marculescu. Modeling and optimization for soft-error reliability of sequential circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(5):803–816, 2008.

Appendices

Appendix A

Python SMT Library of RTL Constructs

This Appendix contains the Python SMT Library of RTL constructs with embedded SEU propagation properties. The signals are represented as a Python List of two bit-vectors. The element 0 of the list represent the logic state of the signal and element 1 of the list represent the fault state of the signal. For example, for a signal **A**, A[0] contains the SMT bit-vector variable representing the logic state of A (i.e. A_l) and A[1] contains the SMT bit-vector variable representing the fault state of A (i.e. A_f).

A.1 Logical AND Operator

```
def and2(A,B):
    logic = A[0] & B[0]
    fault = (A[1]&B[0]&~B[1]) | (B[1]&A[0]&~A[1]) |
                ((A[1]&B[1]) & ~(A[0] ^ B[0]))
    return [logic,fault]
```

Listing A.1: Python Function Defining the Logical AND Operator

A.2 Logical OR Operator

```
def or2(A,B):
    logic = A[0] | B[0]
    fault = (A[1]&~B[0]&~B[1]) | (B[1]&~A[0]&~A[1]) |
              ((A[1]&B[1]) & ~(A[0] ^ B[0]))
    return [logic,fault]
```

Listing A.2: Python Function Defining the Logical OR Operator

A.3 Logical XOR Operator

```
def xor2(A,B):
    logic = A[0] ^ B[0]
    fault = A[1] ^ B[1]
    return [logic,fault]
```

Listing A.3: Python Function Defining the Logical XOR Operator

A.4 Logical NOT Operator

```
def not1(A):
    logic = ~A[0]
    fault = A[1]
    return [logic,fault]
```

Listing A.4: Python Function Defining the Logical NOT Operator

A.5 Reduced AND Operator

```
def redand(A):
    logic = BVRedAnd(A[0])
    c1 = BVRedOr(A[1])
    c2 = BVRedAnd(A[0] ^ A[1])
    c3 = BVRedAnd(A[0])
    fault = c1 & (c2 | c3)
    return [logic,fault]
```

Listing A.5: Python Function Defining the Reduced AND Operator

A.6 Reduced OR Operator

```
def redor(A):
    logic = BVRedOr(A[0])
    c1 = BVRedOr(A[1])
    c2 = BVRedAnd(~(A[0] ^ A[1]))
    c3 = BVRedAnd(~A[0])
    fault = c1 & (c2 | c3)
    return [logic,fault]
```

Listing A.6: Python Function Defining the Reduced OR Operator

A.7 Signed Extend Operator

```
def signExtend(n,A):
    logic = SignExt(n,A[0])
    fault = SignExt(n,A[1])
    return [logic,fault]
```

Listing A.7: Python Function Defining the Signed Extend Operator

A.8 Unsigned Extend Operator

```
def unsignExtend(n,A):
    logic = ZeroExt(n,A[0])
    fault = ZeroExt(n,A[1])
    return [logic,fault]
```

Listing A.8: Python Function Defining the Unsigned Extend Operator

A.9 Concatenation Operator

```
def concat_f(*args):
    logic = args[0][0]
    fault = args[0][1]
    args = args[1:]
    for arg in args:
        logic = Concat(logic,arg[0])
        fault = Concat(fault,arg[1])
    return [logic,fault]
```

Listing A.9: Python Function Defining the Concatenation Operator

A.10 Extract Operator

```
def extract_f(high,low,A):
    return [Extract(high,low,A[0]),Extract(high,low,A[1])]
```

Listing A.10: Python Function Defining the Extract Operator

A.11 Addition Operator

```python
def add(A,B,n):
    A_faulty = A[0] ^ A[1]
    B_faulty = B[0] ^ B[1]
    C_logic = A[0] + B[0]
    C_faulty_logic = A_faulty + B_faulty
    C_fault_state = C_logic ^ C_faulty_logic
    return [Extract(n,0,C_logic), Extract(n,0,C_fault_state)]
```

Listing A.11: Python Function Defining the Addition Operator

A.12 Subtraction Operator

```python
def add(A,B,n):
    A_faulty = A[0] ^ A[1]
    B_faulty = B[0] ^ B[1]
    C_logic = A[0] - B[0]
    C_faulty_logic = A_faulty - B_faulty
    C_fault_state = C_logic ^ C_faulty_logic
    return [Extract(n,0,C_logic), Extract(n,0,C_fault_state)]
```

Listing A.12: Python Function Defining the Subtraction Operator

A.13 Multiplication Operator

```python
def add(A,B,n):
    A_faulty = A[0] ^ A[1]
    B_faulty = B[0] ^ B[1]
    C_logic = A[0] * B[0]
    C_faulty_logic = A_faulty * B_faulty
    C_fault_state = C_logic ^ C_faulty_logic
    return [Extract(n,0,C_logic), Extract(n,0,C_fault_state)]
```

Listing A.13: Python Function Defining the Multiplication Operator

A.14 Division Operator

```
def add(A,B,n):
    A_faulty = A[0] ^ A[1]
    B_faulty = B[0] ^ B[1]
    C_logic = A[0] / B[0]
    C_faulty_logic = A_faulty / B_faulty
    C_fault_state = C_logic ^ C_faulty_logic
    return [Extract(n,0,C_logic), Extract(n,0,C_fault_state)]
```

Listing A.14: Python Function Defining the Division Operator

Appendix B

Python Soft Error Analysis Scripts

This Appendix contains the code of the script for automated SEU injection and propagation analysis. The following code computes the SER. In order to run the script, the Microsoft Z3 Python API [12] must be downloaded and imported. This script computes the propagation probability for every injection site. It also computes and outputs the vulnerability of every output as well sa the value of the Soft Error Rate (SER) of the circuit under test. In order to use the following script, the design must be defined in Python format using our Python SMT library functions.

```python
for o in range(0,n_out): ## For Every Output
    print(o) ## Print Output Under Test
    for i in range(0,n_in): ## For Every Input
        print(simplify(Extract(i,i,inputs))) ## Print Input
        formula = [Extract(o,o,outputs) != 0]
        for j in range(0,n_in):
            if j == i:
                formula.append(Extract(j,j,inputs) == 1)
            else:
                formula.append(Extract(j,j,inputs) == 0)
        print(genModels(And(formula),vr)/(2.0**n_in))
```

Listing B.1: Python Script for SER Computation

In order to use approximate model counting, the tool SMTApproxMC [45] must be downloaded and run separately. The tool only takes as input SMT2 format of SMT formulas. Our library includes a function to convert any given formula into SMT2 format. The resulting output must be saved in a file which can be sent directly into the SMTApproxMC tool [45].

```
def toSMT2Format(f):
    return Z3_benchmark_to_smtlib_string(f.ctx_ref(), "",
            "QF_BV", "", "", 0, (Ast * 0)(), f.as_ast())
```

Listing B.2: Python Function to Convert Formula into SMT2 Format