Providing an Infrastructure for Assertion-based Test Generation and GPU Accelerated Mutation Testing

Jason G. Tong

Department of Electrical and Computer Engineering
McGill University, Montréal, Québec

A Thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical Engineering

December, 2013

The secret in doing anything, is believing that you can do it. Anything that you believe you can do strong enough, you can do it. As long as you believe.

— Bob Ross

You've gotta hit as hard as life. It ain't about how hard you hit, it's about how hard you can get hit and keep moving forward. How much can you take and keep moving forward. That's how winning is done!

— Rocky Balboa

Acknowledgements

Fourteen years ago, I had great aspirations of pursuing a Ph.D. degree in Electrical Engineering. Fourteen years later, I successfully did it. And it is a grand honour and a privilege of achieving this at McGill University.

There are several people I want to thank. First and foremost, I dearly thank my supervisor Dr. Zeljko Zilic and Dr. Marc Boulé. I thank you both for your invaluable guidance and assistance over the last 6 years. I am also thankful to you both for introducing to me to Assertion-based verification and GPU computing. I am greatly indebted for their invaluable feedback, especially with the contributions by Marc with our published papers and journal articles. Without their help, this research would not have been possible.

Next, I want to thank Dr. Yann Oddos, for working together in Assertion-based test generation and also for his generosity lending his *MyGen* tool for my research. I want to acknowledge the verification team at PMC-Sierra, particularly Patrick Tsinany, Dr. Ken Wagner and Claude Beauregard of Cadence Design Systems of Canada with their time and patience in helping me to get started with Cadence Incisive Formal Verifier. I also thank them for their generous donation of their assertion and design benchmarks that were used in this thesis. Additionally, I would like to acknowledge Darcy Poncsak, Kevin Peterson, and Ricardo Nunez for helping me to get settled into my very first engineering job and made my stay at PMC-Sierra enjoyable.

I would like to acknowledge Kelly Goss from Acceleware for her valuable GPU computing and CUDA training. Thank you for enlightening me with some ideas, which were incorporated into GPU kernels that were developed in this thesis. Also, to Olive Zhao and the Canadian Microelectronics Corporation for their generous donation of the NVIDIA GPU Tesla cards and workstations.

To Kenneth Domino, thanks for our lengthy and thorough discussions on helping me to get started with ANTLR and your invaluable ANTLR C++ parser program, which was extensively used with the development of my tools of this research.

To the members of the Integrated Microsystems Laboratory (in no particular order) I want to thank, Omid Sarbishei, Majid Janidarmian and Atena Roshan Fekr for their insightful and inspirational research discussions that helped me in developing the tools in this thesis. Additionally, I want to acknowledge Marwan Kanaan, Pang Yu, Rozita Najafi-Nejad, Ashraf Suyyagh, Ari Ramdial, Ben Nahill, Steve Ding, Dimitrios Stamoulis, Alexandre Courtemanche and Dr. Andraws Swidan, for making my stay in the lab fun and enjoyable. Also, to Connie Greco, I thank you for helping

me with the administrative work, particularly with the thesis submission process and setting up my defence, which required a lot of your time.

I also want to thank the people from the McMaster University, particularly Dr. Nicola Nicolici, and his former Computer Aided Design and Test group members (in no particular order): Ehab Anis, Adam Kinsman, Henry Ho Fai Ko, Zahra Lak, Mark Jobes, Kaveh Elizeh, Jason Thong and Roomi Sahi. I greatly thank you all for helping me to find my *niche and passion* for software development, which has helped me immensely in completing this research. I also want to thank Cheryl Gies for helping me with the immense administrative work, and who I am very glad that I met during my brief stay.

To my closest friends: Natalia Salgo, Sheeba Pennickara, Lisa and Sandra Price, Frenita Chua, Frances Chua, Shan-Shan Chua, Mr. Chen, Noman Siddiqui, Dr. Mohammed A. S. Khalid, Laura Ciovica, Courtney Padden, Courtney Beaulieu, Jasmin Pichlyk, Savannah Garoufalis, Jessica Perez, Ayo Olanrewaju and Cassandra Stover. Thanks so much for your support and friendship over the years.

Last but not least, I want to thank my parents for their love, support and encouragement. I am greatly indebted for the sacrifices that they made just for me to pursue and fulfill my dream.

Abstract

Functional verification of modern digital designs is a never ending challenge in the Integrated Circuit (IC) industry. Fuelled by the continuous demand of more integration, the increased effort in verification does not always entail error-free circuits after first production. Emerging technologies such as Assertion-based verification, can help in verifying the functional correctness of digital designs and can be easily integrated into existing design verification methodologies. Simulation-based verification is still the most predominant method in industry because of its ability to scale with large designs. Assertions can be inserted into the design and they can be treated as coverage points, where the input tests are responsible for exerting the design's conditions in evaluating those assertions. The effectiveness of this approach relies on the quality of the tests, where poor test quality can prevent the design from being thoroughly verified.

This thesis presents novel techniques and algorithms for generating tests from assertions. Assertions serve as an invaluable source of information, where one can leverage the defined behaviours for generating the appropriate functional tests that can be used in simulation. A proposed set of coverage metrics helps in generating tests that thoroughly evaluate assertions during simulation. Verification engineers can make use of these tests in performing effective simulation in order to detect and then correct any design errors. The tool developed for generating tests from assertions was evaluated using nearly 300 assertions that were written for verifying the correctness of several industry-based designs. As a result, the proposed test generation approach was able to provide additional tests which led to an improvement in coverage compared to assertion-based test generator developed by another research team.

Mutation testing is a technique that can be used for gauging the quality of assertion-based tests. This thesis also developed novel algorithms for Graphics Processing Units and used for accelerating mutation-based simulations, which is a computationally intensive application. It was empirically shown for a set of 10 industry-based designs, that efficiently using the GPU's resources can drastically improve the simulation performance on the GPU, when compared to a commercial tool. The additional performance is a necessity, where maximal acceleration is needed for rigorously assessing test quality when simulating large quantities of mutations. This can have a positive impact in the quest for improving assertion quality, ultimately leading to an effective dynamic verification of digital designs.

Abrégé

La vérification fonctionnelle de circuits numériques modernes comporte des défis sans fin dans l'industrie des circuits intégrés (CI). Alimentés par la demande continue d'intégration croissante, les efforts grandissants en vérification ne mènent pas toujours à des circuits sans erreur du premier coup. Une technologie émergente telle que la vérification par assertions peut aider à vérifier le bon fonctionnement des circuits numériques et peut être facilement intégrée aux méthodologies de vérification existantes. La simulation fonctionnelle représente toujours la méthode de vérification la plus répandue dans l'industrie, étant donné sa capacité à traiter des circuits plus volumineux. Les assertions peuvent être insérées dans un circuit et peuvent aussi servir comme repères de couverture, pour lesquels les tests d'entrée ont la responsabilité d'exercer le circuit évaluant ces assertions. L'efficacité de cette approche repose sur la qualité des tests, car de piètres tests peuvent empêcher une vérification complète.

Cette thèse présente des techniques et algorithmes novateurs ayant pour but de produire des tests à partir des assertions. En raison des comportements qu'elles décrivent, les assertions représentent une source importante d'information permettant d'extraire des séries de tests fonctionnels, pouvant servir lors de la simulation. Un ensemble de métriques de couverture aide à produire des tests qui évaluent rigoureusement les assertions durant la simulation. Les ingénieurs en vérification peuvent ainsi utiliser ces tests pour effectuer des simulations efficaces dans le but de détecter et corriger des erreurs de conception. L'outil servant à générer des tests à partir des assertions qui a été développé fut évalué avec près de 300 assertions créées dans le but de vérifier le bon fonctionnement de plusieurs circuits industriels. Sur le plan des résultats, l'approche de génération de test proposée a été capable de produire des tests supplémentaires menant à une couverture de test améliorée comparativement à un générateur de test d'une autre équipe de recherche.

Le test par mutation est une technique permettant d'évaluer la qualité des tests découlant des assertions. Les simulations de mutations exigent une grande puissance de calcul. Basés sur des processeurs graphiques (GPU), cette thèse présente aussi des algorithmes novateurs dans le domaine des tests par mutations. Sur une série de 10 circuits industriels, les résultats expérimentaux démontrent une amélioration importante de la performance de simulation comparativement à un outil commercial. Cette amélioration des performances est nécessaire étant donné l'accélération de calcul requise pour évaluer la qualité des tests lors de simulations de plusieurs mutations. Cela a un impact bénéfique dans la quête visant à améliorer la qualité des assertions, menant ultimement vers une vérification dynamique efficace de circuits numériques.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

μ-GSIM :	Mutation Simulator on GPU
μDV-GSIM :	Mutation-based Dynamic Verification Simulator on GPU
Airwolf-TG :	Airwolf Test Generator
Airwolf-CTG :	Airwolf Compacted Test Generator
ABD :	Assertion-Based Design
ABV :	Assertion-Based Verification
AB-DV :	Assertion-based Dynamic Verification
AHB :	AMBA's Advanced High-performance Bus
AXI :	AMBA's Advanced eXtensible Interface
AM :	Assertion Map
ATPG :	Automatic Test Pattern Generation
ATR :	Additional Test Reduction
BFS :	Breadth-First Search
CPU :	Central Processing Unit
CPF :	Circuit Parallelism Factor
CRTC :	Complete Round Trip Coverage
DUT :	Design Under Test
DUV :	Design Under Verification
DFS :	Depth-First Search
EDA :	Electronic Design Automation
FI :	Fault Injection
FF-SYNC :	FIFO Synchronous Controller
FF-GC :	FIFO Grey Code Controller
FPGA :	Field Programmable Gate Array
GPU :	Graphics Processing Unit

HSA :	Hybrid Search Algorithm
HDL :	Hardware Description Language
IC :	Integrated Circuit
IEEE :	Institute of Electrical and Electronics Engineers
IP :	Intellectual Property
LFSR	Linear Feedback Shift Register
MAE	Multiple Assertion Encoding
MBAC:	Marc Boulé's Assertion Compiler
MC :	Model Checking
MFG :	Multiple Fault Gate Encoding
MI :	Mutant Injection
MMG :	Multiple Mutant Gate Encoding
NFA :	Non-deterministic Finite Automaton
NSA :	Node Selection Algorithm
OpenCL :	Open Computing Language
OTR :	Overall Test Reduction
PCI :	Peripheral Component Interconnect
PPR :	Parallel Path Removal Algorithm
PSL :	Property Specification Language
RTL :	Register Transfer Level
SAE :	Single Assertion Encoding
SFG :	Single Fault Gate Encoding
SMG :	Single Mutant Gate Encoding
SDRAM :	Synchronous Dynamic Random Access Memory
SVA :	System Verilog Assertions
TPO :	Test Path Overlap Algorithm
VHDL :	Very High Speed Integrated Circuit Hardware Description Language

Chapter 1

Introduction

This chapter presents an overview of the thesis, along with a description of the problem definition. Also, a summary is given on contributions that were brought forth into this thesis.

1.1 Problem Definition and Motivation

Function verification of modern digital designs is a never ending challenge in the Integrated Circuit (IC) industry. Fuelled by the continuous demand of electronic devices comprising many integrated components, companies have invested a significant amount of resources into functional verification. Nowadays, verification can consume as much as 70% of the total development of a product [3]. Companies must strive to reduce the number of design errors that can escape prior to mass production. Producing products containing undetected design errors can be very costly. Not only the company's market share will be lost to the competition, human lives can be at stake. Our modern society is technologically driven and reliant on these digital systems. Thus, it is increasingly important to ensure electronic devices are free from functional errors.

Figure 1.1 shows the data of the collected survey responses from various IC companies that was conducted by the Wilson Research Group and Mentor Graphics [1]. IC companies have reported that *functional and logic* bugs are most prevalent during product development and verification. Additionally, the survey also reported that as much as five re-spins are needed due to functional bugs escaping during verification [1]; however, despite these costly re-spins, design errors do go undetected, which then become part of the product that is mass produced. For instance, in 1994, Intel's infamous FDIV bug was manufactured into their Pentium brand of processors. Massive

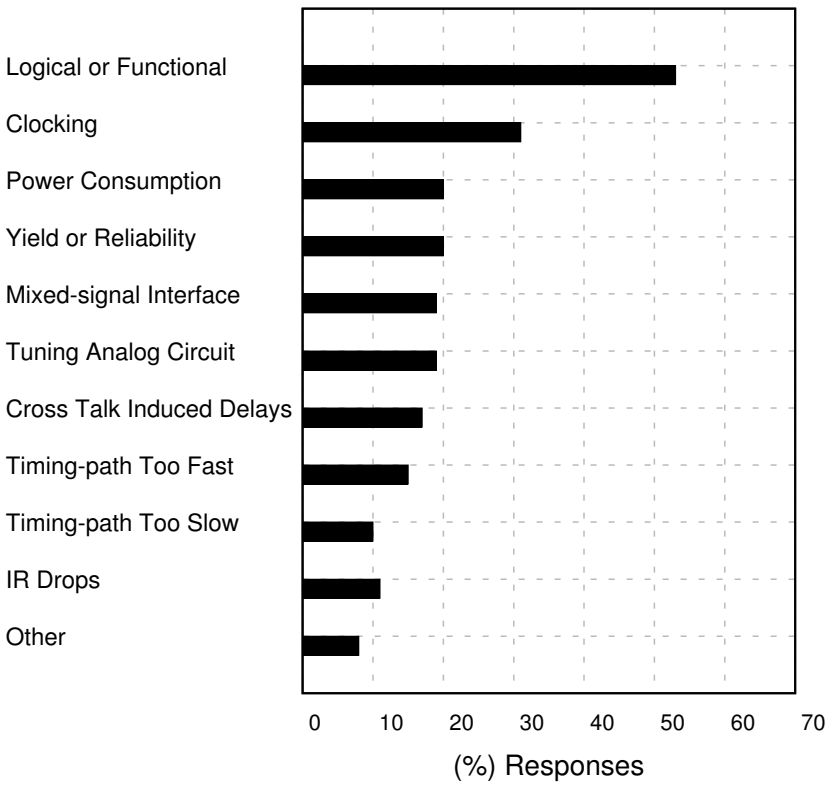


Figure 1.1: Common Bugs During IC Design and Verification [1]

recalls were needed to replace these processors containing the floating point division error. The cause of the error was not a result during manufacturing, but a functional fault that was undetected during verification. The total cost in replacing and fixing the bug was nearly 480 million dollars. More recently in 2007, Advanced Micro Devices Phenom microprocessors suffered the Translation Look-aside Buffer functional bug, which was discovered weeks after mass production. This bug was also a functional error, which can cause system instability during operation. A software patch was released as a workaround of the flaw; however, this caused significant performance penalties. Both of these examples show the importance of functional verification in which errors going undetected can have drastic economical effects to the company.

Recently, Assertion-based Verification (ABV) has begun a prominent aspect in functional verification [4]. Assertions are high-level statements that are capable of describing the correct behaviour of the design, with the intent of capturing any design errors. Assertions can be derived from a set of specifications, which describes the requirements to which the design must conform. A typical design requirement specifies the expected response that the design must generate based on some input conditions. Any deviation in the design's output response causes an assertion to fail,

which this result can be computed by means of simulation or formal methods.

Assertions can be inserted into the design and simulated with a digital circuit simulator or formally checked with a model checker. As designs become enormously large, model checking becomes infeasible due to its state space explosion issues. Simulation is the most predominant method in industry because of its ability to scale with larger designs. Incorporating assertions into simulation can further help to assess how well the design has satisfied the specification. Each assertion describes an aspect (or a required functionality) of the design. During simulation, the simulator tool captures every activity within the design caused by the set of tests that is applied at its inputs. Then, the recorded activity determines which assertions were satisfied (or falsified); however, the verification thoroughness depends on the quality of the tests that are used during simulation.

Assertions not only monitor the behaviour of the design, but they can also be used as a valuable source of information for generating appropriate input stimuli (tests). Assertions can serve as a *blueprint*, where the defined sequence of events that is specified in an assertion, is used for generating the appropriate functional tests for creating an assertion pass. It is important to note that throughout this thesis, the word "test" is specifically used for detecting design errors and should not be confused with detecting manufacturing faults. Automated test generation from assertions is advantageous compared to writing directed tests manually. An assertion can have complex temporal sequences that is mixed with Boolean expressions which define a required condition that the design must satisfy. This can potentially create an enormous amount of tests and performing a manual test generation would be infeasible.

The work presented in this research lies in the area of automated test generation. The intent is to develop an automated technique for generating tests from assertions, which can be used for dynamically verifying the functional correctness of a design. The effectiveness of simulation-based verification is reliant on the test quality. Poor test quality can lead to portions of the design being unexplored, thereby having potential bugs go undetected. Since assertions can be inserted into the design during simulation, the generated assertion-based tests can then be applied to the primary inputs. The assertion-based tests will effectively exert the necessary conditions that were defined in the assertion itself. This will help in quickly evaluating in how well the design has implemented the specification, which is modelled by the assertions. This is beneficial when compared to using pseudorandom functional tests.

Another contribution that was made in this work, is to devise a method in har-

nessing the raw compute power of Graphics Processing Units (GPUs) for accelerating mutation-based simulations. In ABV, mutation testing relies on simulating many mutated designs individually, with the intent of gauging the quality of assertion-based tests that are able to uncover the injected mutant (functional fault). GPUs are an ideal hardware platform because simulating multiple mutated designs can be performed independently. This is favourable when modern digital designs are injected with large quantities of mutations.

1.2 Problem Definition

Incorporating ABV into dynamic verification comes with challenges that have to be addressed. A common problem when using assertions is vacuity. Passing a set of assertions may not necessarily imply that the design is entirely bug-free. An assertion can have more than one sequence of conditions that results in an *assertion pass*. Additionally, an assertion pass can also occur when the assertion itself was not evaluated during verification. This happens when the design is unable to exert all the necessary conditions that are needed for the assertion to begin monitoring its behaviour. When the assertion was not thoroughly exercised, then it has experienced a *vacuous pass* [5].

Another problem is to devise an accelerated method for assessing the test quality from assertions. Despite the ability of assertions on specifying design intent, they can also generate tests that may not necessarily exert the entire design, which can lead to poor test quality. Mutation testing is a common method for assessing the quality of tests. A design is injected with a set of mutations (functional faults), where each mutated design is simulated using the assertion-based tests; however, the large computational costs that is accompanied with mutation testing makes it impractical to use, which is due to simulations of many mutated designs.

Given the importance and the challenges of ABV and mutation testing, this thesis will focus on the following two problems:

1. *How can one know when the test set is effectively exercising the assertions describing the expected circuit behaviour?*
2. *How to harness the raw compute power of Graphics Processing Units for accelerating the simulation of multiple designs?*

The first challenge was addressed in Chapter 3 where a set of coverage metrics were defined for generating tests using Non-deterministic Finite Automata (NFA)

representations of assertions. These defined coverage metrics are incorporated into the proposed test generator tool for assertions, which undertakes *coverage driven test generation* for producing tests that non-vacuously pass (or fail) an assertion. The proposed coverage metrics that are incorporated into the test generator have contributed to an additional test coverage when compared to another assertion-based test generator tool. Assertion-based test compaction was also explored in this research. The purpose is to cluster similar assertions into groups and attempt to exploit similarities of other assertions that can be shared with each other. Two novel test compaction algorithms are described in Chapter 4, which use similarities in identifying any redundant test paths that can also be satisfied from another assertion. Ultimately, this is beneficial in reducing the applied verification time.

The second challenge was addressed in Chapters 5 and 6. GPUs are massively parallel devices that are suitable for accelerating many forms of computation, such as bioinformatics, medical imaging and digital circuit simulation. GPUs can also help in accelerating circuit simulation of large designs. The key to harness the raw computer power of GPUs is to ensure the circuit data are independent of each other. This has led to the development of a novel data-generation algorithm for exploiting data-parallelism (more specifically bit-parallelism) of the circuit design. The entire length of GPU's computer word was leveraged for storing different kinds of mutant and fault information, along with assertion-based test data. This has led to an innovative data encoding scheme that was used for generating efficient data-parallel representations of multiple circuits. The work presented in Chapter 6 also uses similar encoding techniques, but instead assertion-based tests were encoded over the computer word. It was empirically shown from the developed encoding techniques that memory efficiency is key in improving simulation performance.

1.3 Thesis Contributions and Collaborations

This section presents a summary of the contributions that were made from this research. The author would like to highlight the contributions that were also made from external collaborators within the Integrated Microsystem Laboratory and PMC-Sierra, which are described in detail. These contributions are organized by the presented topics within this thesis.

- *Coverage Driven Assertion-based Test Generation*: A set of coverage metrics was proposed when using the NFA as the *computable* representations of assertions.

The NFA representations are generated by the hardware assertion checker tool, *MBAC*. The coverage metrics serve as a guide in generating tests from NFAs which includes all the specified conditions at least once, without having any condition unexplored. These coverage metrics were then incorporated into the test generator tool, *Airwolf-TG*, which helps in producing tests that can non-vacuously pass (or fail) an assertion. Using a set of industrial assertions that were presented from [2], has shown additional tests were generated for improving the overall assertion coverage, by as much as 70% in some of the benchmarks.

The author developed heuristic algorithms for generating tests using NFA representations of assertions. This led to a developed novel *Hybrid Search Algorithm*, which incorporates the proposed coverage metrics for generating assertion-based tests non-vacuously. Also, this work was performed in collaboration with M. Boulé who assisted in the translating of the PSL assertions that were used from *MyGen*[2]. M. Boulé also provided the *MBAC* tool[6] for generating NFA representations of various assertions. Y. Oddos also generously provided the *MyGen* tool, which is used for generating test sequences from their hardware generators. This has also led the author to develop an assertion coverage analyzer, *Airwolf-CA*, which assessed the assertion coverage of the tests that were generated from *MyGen*. This work has resulted in the following publications:

- **J. G. Tong**, M. Boulé and Z. Zilic, *Defining and Providing Coverage for Assertion-based Dynamic Verification*[1] [7] Journal of Electronic Testing: Theory and Applications (JETTA), Special Issue on the High-level Design Verification and Test workshop, Volume 26, Issue 2, pp. 211–225, April 2010
- **J. G. Tong**, M. Boulé and Z. Zilic, *Airwolf-TG: A Test Generator for Assertion-based Dynamic Verification* [8], In the Proceedings of the 14$^{\text{th}}$ High-Level Design Verification and Test workshop (HLDVT'09), pp. 106–113, November 2009

In addition to this work, the coverage analyzer was also used for analyzing a set of assertions for verifying an industrial FIFO controller. This work was performed at an internship sponsored by *PMC-Sierra* and a *Natural Sciences and Engineering Research Council Engage Industrial Grant*. The FIFO controller and set of assertions were developed in-house and were gladly provided

[1]This publication was selected as an **Invited Journal Article**

by P. Tsinany and K. Wagner. The coverage analyzer also used assertions that were developed by D. Sarraf. The author was responsible for extracting tests from a formal verification tool, *Cadence Incisive Formal Verifier*, and used the developed coverage analyzer in gauging how well the tests exercised all the specified conditions within each assertion. This work has resulted in the following publication:

- **J. G. Tong**, D. Sarraf, M. Boulé and Z. Zilic, *Generating Compact Assertions for Control-based Logic Signals*, In the Proceedings of the 54th Midwest Symposium on Circuits and Systems (MWSCAS'11), pp. 1–4, August 2011

- *Assertion-based Test Compaction*:

  Two test compaction algorithms were developed by the author, which use clusters of similar assertions for the intent of reducing the size of assertion-based tests. A clustering algorithm was devised, which groups similar assertions with a certain level of similarity. One of test compaction algorithms leveraged the similarities of the test paths from different assertions. The purpose is to identify any redundant sequences of events from different assertions that can be subsumed with each other. An improved test compaction algorithm attempts to assign multiple specified conditions from many assertions within a single test. The proposed test compaction approaches were evaluated using nearly 300 industry-based assertions, and have showed a visible test set reduction by as much as 98% in some of the benchmarks.

  The author would like to highlight the contributions made by M. H. Neishaburi who assisted in generating the NFA representations of the AXI benchmark [9]. Also, a thorough discussion was made on how to cluster assertions, where M. H. Neishaburi's work was used for clustering hardware checkers while the author's work was to cluster the assertion (statements) themselves. Constructive discussions were also made with O. Sarbishei about the MiniSAT [10] solver, which was helpful for integrating it into *Airwolf-CTG*. Additionally, this has also helped in the development of the two test compaction algorithms. This work has resulted in the following publications:

  - **J. G. Tong**, M. Boulé and Z. Zilic, *Test Compaction Techniques for Assertion-based Test Generation* [11], In the ACM Transactions on De-

sign Automation of Electronic Systems (TODAES), Volume 19, Issue 1, pp. 1–29, December 2013

- **J. G. Tong**, M. Boulé and Z. Zilic [12], *Assertion Clustering for Compacted Test Sequence Generation*, In the Proceedings of the 13th International Symposium on Quality Electronic Design (ISQED'12), pp. 694–701, March 2012

- *GPU Mutation and Fault Simulation:*

  A data-parallel generation algorithm was developed for the purpose of performing mutation and fault simulation of digital circuits using GPUs. Two novel data-parallel representations were proposed, namely *mutant* and *gate* streams. Each proposed representation exploited several levels of parallelism, namely gate, mutant and and fault parallelism, for the purpose of having one work-item to simulate many mutated and fault-injected circuits. The developed data generation algorithm made use of the GPU's computer word for encoding different gate, fault and mutant data. This has further led to two proposed encoding schemes, namely Multiple Mutant Gate (MMG) and Multiple Fault Gate (MFG) encoding, which were used in mutation and fault simulation on the GPU respectively. These encoding schemes compactly store gate, mutant and fault information over the computer word. Additionally, mathematical formulations were proposed in determining the optimal number of work-items that can be launched on the GPU which is based on memory usage. The combined parallelism factors gave a noticeable reduction of the memory usage of the circuit and mutant data. This allowed for the host program to allocate more work-items on the GPU, thereby improving simulation efficiency. It was empirically shown that reducing the memory usage for simulating 8 industrial benchmarks, has led to an improvement in simulation performance by at least 95× in mutation simulation, and 26× in fault simulation.

  The author would like to highlight the many discussions that were had with M. Boulé who has suggested the idea of using gate-level mutations. This helped in the development of a novel encoding scheme used for storing necessary gate-level information over the GPU's computer word. Also, this work was extended into fault simulation for manufacturing testing. This work has resulted in the following publications:

  - **J. G. Tong**, M. Boulé and Z. Zilic, *Improving GPU Fault Simulation*

with Efficient Data-parallel Generation Techniques, IEEE Transactions on Computer Aided Design, (In Submission – January 2014)

- **J. G. Tong**, M. Boulé and Z. Zilic, *Efficient Data Encoding for Improving Fault Simulation Performance on GPUs* [13], To appear in the 4th International Symposium on Electronic System Design (ISED'13), December 2013
- **J. G. Tong**, M. Boulé and Z. Zilic, *Mu-GSIM: A Mutation Testing Simulator on GPUs*[2] [14], In the Proceedings of the 5th Asia Symposium on Quality Electronic Design (ASQED'13), pp. 302-311, August 2013

Additionally, this work was extended for performing mutation testing in the context of design verification using assertions. This extended work led to a development of a novel encoding scheme, namely Multiple Assertion Encoding (MAE). The proposed encoding scheme differs from the MMG and MFG encodings. The MAE compactly stores multiple tests from different assertions over the GPU's computer word. This allows for different assertion-based tests within one work-item to be simulated over large quantities of mutated designs on the GPU. MAE further reduced the memory usage on 3 industrial designs, which led to an improvement of the simulation efficiency by as much as 10× over an RTL simulator. Accelerating mutation testing is greatly beneficial, where assertions are rigorously assessed when using large quantities of mutations. This work has resulted in the following publications:

- **J. G. Tong**, M. Boulé and Z. Zilic, *Using GPUs for Accelerating Mutation Testing for Design Verification*, IET Computers & Digital Techniques, (In Submission – January 2014)

The work listed in the contributions were performed by the author, who was solely responsible in the implementation and development of test generation and GPU simulation algorithms.

The author performed and analyzed all the experimental results of each of the proposed tools in this thesis. These contributions also came from the continuous collaborative work with M. Boulé. It is noted that M. Boulé has provided invaluable feedback and also contributed to the technical writing of these papers. Z. Zilic supervised the entire work.

[2]This publication was awarded an **ASQED'13 Best Paper Award**

The organization of the entire thesis is as follows: Chapter 2 presents the necessary background that is used in this thesis. Section 2.4 presents a brief survey of the previous work related to assertion-based test generation, test compaction, GPU logic and fault simulations and mutation analysis of assertions. Chapter 3 introduces *Airwolf-TG*, the coverage driven test generator for assertion-based dynamic verification. Chapter 4 presents the proposed test compaction methodology for assertion-based tests which is incorporated inside the test compactor, namely *Airwolf-CTG*. Chapter 5 presents a comprehensive description of the data encoding technique for mapping multiple mutant and fault data on the GPU's device memory. Then the two GPU-based simulators are introduced, namely *μ-GSIM* (Section 5.2) and *GS-SIM* (Section 5.3) respectively. Finally, Chapter 6 presents μDV-GSIM which utilizes these efficient data encoding techniques and performs mutation testing for assessing the quality of assertions.

Chapter 2

Background and Related Work

This chapter begins with describing the theme of this work within the area of functional verification using assertions. Then, a brief introductory background on assertions is presented along with the definitions that are used throughout this thesis. Following is a discussion on mutation testing, which is used for measuring assertion quality. Subequently, an overview of the Graphics Processing Unit is shown with a brief discussion on OpenCL are given. This chapter ends with a survey of the related research work that is related to assertion-based test generation, test compaction, GPU-based logic and fault simulation, and applying mutation testing on assertions.

2.1 Functional Verification with Assertions

Functional verification of Integrated Circuits (IC) investigates if the design is thoroughly verified in terms of its correct operation and functional completeness [15]. Two different terms that often arise when testing and verifying hardware designs, which are: *validation* and *verification*. *Validation* seeks to determine if the design meets the intended use and satisfies the customer's requirements. *Verification* confirms if the design was properly built and if it reflects the design's specifications [16]. The ultimate goal is to detect and correct any errors that may exist in the design.

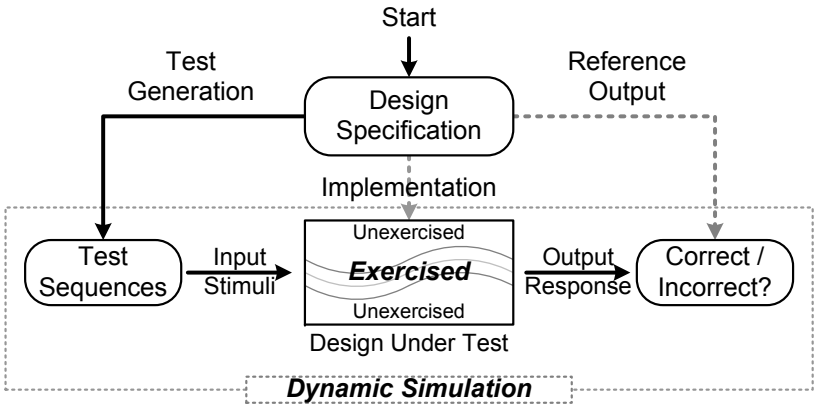
As designs become more complex due to the increase by customer demand for more functionality, integration and performance, verification becomes a crucial phase during development. IC design companies have invested significant resources in verification; however, this does not always entail in producing error-free designs after first production. IC companies constantly struggle with the increasing verification gap [17], where the complexity of digital designs is significantly greater than verification abilities. More design errors escape during the test and verification phase in which

can lead to non-functional designs being produced. This can put a major strain on the company's resources in performing costly re-spins of the design. Additionally, this can further delay the development progress and potentially lead to missed time-to-market deadlines. Thus, it is imperative that designs are bug-free prior to production.

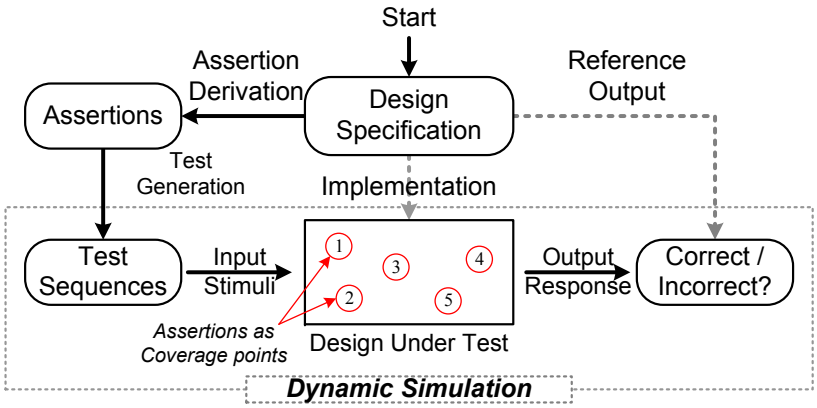
Assertion-based verification has swept into the field of design verification and has begun to play a prominent role in functionally verifying the correctness of digital designs [3]. Assertions are high-level statements, armed with temporal logic and Boolean operators, which are able to describe a broad range of expected (or prohibited) behaviour of the design. They are derived from a set of specifications that describe the correct functionality of the design. Recently, verification groups and the Electronic Design Automation (EDA) industry have incorporated assertions into their verification methodology and into verification tools. This has led to the emergence of two leading industry standards, namely the Property Specification Language (PSL) [18] and the SystemVerilog Assertion [19] languages.

There are two categories of verification using assertions for functionally verifying the correctness of a design, namely *static* and *dynamic* verification. The goal in both methods is to find and correct any bugs that may exist within the design. *Static verification* (or formal) transforms the design into an abstract model, which depicts its functionality. A set of mathematical formulas, in terms of temporal logic, define the intended (or prohibited) behaviour. These formulas are the assertions that depict the properties to which the design must conform. A formal verification tool (such as model checkers [20] or theorem provers) exhaustively checks the model in determining if the properties are proven (or dis-proven) correct. Whenever a property (assertion) passes, the model checker generates a *witness trace* that describes the sequence of events that satisfies the assertion. Otherwise, a *counterexample* is produced, depicting the events leading to an assertion failure. Static verification is known for its capacity limitations, which limits the size of the design that can be verified when using formal verification tools.

Dynamic verification (or simulation) is the most predominant method in industry for validating the correctness of a design due to its scalability. Simulation-based verification typically comprises of the design itself and a set of *functional tests* that is used for input stimuli. (It is important to note that this should not be confused with tests used for manufacturing testing). The design contains a set of assertions used for verifying its functional correctness. During simulation, the simulator tool monitors the assertions and analyzes the simulation traces in determining if any assertion has passed or failed; however, the effectiveness of this approach is highly dependent on



(a) Dynamic Verification



(b) Assertion-based Dynamic Verification

Figure 2.1: Traditional and Assertion-based Dynamic Verification

the quality of the functional test sequences that are applied to the design and on their ability to exert the circuit's functionality for uncovering potential design errors. Functional test sequences can be obtained from various sources, namely from pseudo-random, constrained-random and directed test generation methods. Directed, or algorithmic, test generation can achieve smaller test size and higher coverage compared to random generation approaches; however, directed tests can still produce a large test volume and be time-consuming to develop when performed manually.

2.1.1 Assertion-based Dynamic Verification

In this thesis, assertion-based dynamic verification is used for input test stimuli and also for measuring the quality of assertions. Figure 2.1 shows the differences between traditional and assertion-based dynamic simulation methods. Traditionally, dynamic

Figure 2.2: Assert and Assume Directives

simulation involves a Design Under Test (DUT) and a set of tests sequences that can be pseudo-randomly generated from a test generator tool or a testbench, shown in Figure 2.1(a). The test sequences are applied at the DUT's inputs while the simulator monitors the outputs. The outputs that were generated by the DUT determines if the design was able to exert the correct behaviour as defined from the design specification.

Assertion (or Property)-based Dynamic Verification (AB-DV) [21] treat assertions as *coverage points*, which are indicated by the red nodes in Figure 2.1(b). Assertions are derived from the design specification and are used to describe the required functionality of the design. Additionally, they serve as a valuable source of information such that the defined sequence of events is used for generating the tests required to evaluate the assertion. When the assertion-based tests are applied to the DUT, the DUT will attempt to exert the necessary conditions in order to satisfy (cover) the assertion [22]; however, the effectiveness of this approach is limited to how well the tests are able to exert the necessary conditions for reaching those coverage points.

One of the challenges in verification is that the DUT can have components that are supplied from different vendors which do not necessarily reveal their internal structure. Verifying the functional correctness of these components is then limited to the given specifications where the assertions can only reference the signals at the interface (primary inputs and outputs). A key feature of assertions is that they can also be used for providing the assumptions about the environment, which helps in restricting the input valuations that the component can receive. Figure 2.2 shows the differences when using ϕ with the `assert` and the `assume` directives. The `assert` directive is shown in Figure 2.2(a) monitors the outputs of IP Block A for an assertion defined as, ϕ. Based on the output values, `assert(`ϕ`)` determines if the assertion passes or fails. The same sequence of events that are defined in ϕ can also be used with the `assume` directive, shown in Figure 2.2(b). The statement `assume(`ϕ`)` provides the verification tool the assumptions about the environment of IP Block B, which restricts the kind of input values that can be applied at the inputs. The distinction between using ϕ as an assertion or an assumption is key to the *assume-guarantee paradigm*, where

the outputs of IP Block A are first proven correct by the `assert` directive, then the functionality of IP Block B can also be proven correct when connected to the outputs of IP Block A [23, 24]. This concept helps verification engineers to verify components separately, thereby undertaking a *black box* modular verification approach. This principle is used for generating tests from assertions where the sequence of events and Boolean signals reference the primary inputs of the DUT.

Using a white box testing approach allows access to the DUT's internal structure and provides insight on deriving assertions that verify the correctness of the design. A model checker is employed where the DUT along with the assertion are used for generating the appropriate tests. The typical goal of white box testing is to verify the correctness of the internal implementation and does not incorporate the DUT's environment. Employing a black box approach eliminates the need for the DUT's internal structure, where assertions reference the primary inputs and outputs. The emphasis is to functionally verify the correctness of the DUT to ensure it can produce the correct result based on the given input scenarios defined in the assertions. This helps to limit the search space for generating tests from an intermediate representation of the assertion. Using black box testing is restricted to the access of the primary inputs and outputs, and it can be difficult to derive tests and assertions in verifying the DUT if the specification is not given.

In this thesis, the black box testing approach is undertaken for generating functional tests from assertions. It is assumed that the given assertions describe the functionality of the design by referencing the primary inputs and outputs of the DUT. The *assume guarantee paradigm* allows the use of the defined sequences from assertions as a source for test generation, where these tests create an assertion pass or failure. The next section describes an overview of the SystemVerilog Assertion syntax and followed by a discussion of using a *computable representation* for generating tests from assertions.

2.1.2 Overview of Assertions

Assertions are capable of expressing a broad range of expected (or prohibited) behaviours of the Design Under Test (DUT). Verifiers can take advantage of the expressive power provided by the modern assertion languages such as the *Property Specification Language* (PSL) [18] and *SystemVerilog Assertions* (SVA) [19] for writing complex properties comprised of a wide range of temporal and Boolean operators. At their lowest level, they describe the behaviour by defining the expected values of

different signals in the form of a Boolean expression, b_i. This is also known as the *Boolean Layer* of the assertion language. Each b_i can be a signal itself or a Boolean expression over multiple signals. Assertions usually combine different Boolean expressions together in order to form a sequence of Boolean events. A *sequence* s_i is a finite list of Boolean expressions that describes the precise order of the expected values of signals that must take place over a period of time. Every assertion is defined with a default clocking directive so that each specified Boolean event must be observed at every edge of the clock signal. For instance, the following sequence $s_1 = \{b_1, b_2, \ldots, b_{n-1}, b_n\}$ is said to *match* if and only if each Boolean expression is evaluated to true in n successive clock cycles. Sequences can also be constructed using regular expression operators and other syntax sugaring. Together, these operators comprise the expressive power of the *temporal layer* provided by modern assertion languages.

Properties add another layer of operators that define how these Boolean expressions and sequences should behave. There are many operators and forms available for writing a property; however, this thesis focuses on the following form:

$$\phi_i : \texttt{sequence\_expression |=> property\_expression}$$

where ϕ_i represents an assertion that is used to describe one particular aspect of the DUT's behaviour. This is a typical form that is usually seen in the vast majority of industry-based designs.

The `|=>` operator represents non-overlapping implication. It specifies that the assertion is conditionally checked: if the left hand side prerequisite `sequence_expression` is matched, then the right hand side `property_expression` must also match. These are referred to as the *antecedent* and *consequent* components of the assertion respectively. A failure to observe the consequent when the antecedent has occurred means that the property does not hold on the design.

Definition 1 (Antecedent) *The antecedent of an assertion defines the condition in the form of a sequence of Boolean events that must occur in order for the assertion to evaluate the property in the consequent component.*

Definition 2 (Consequent) *The consequent of an assertion defines the expected sequence of Boolean events that be must be generated by the DUT after a matching sequence from the antecedent component.*

The SVA language is used for showing how the above constructs and different layers are applied; however, the themes presented can also be applicable to PSL

assertions. Consider the assertion ϕ_1, which specifies the behaviour of a certain aspect of the DUT.

ϕ_1: `sig1 ##1 sig3 |=> sig2 ##1 sig5`

The antecedent of assertion ϕ_1 specifies that the gating condition requires signal `sig1` to be at logic-1 followed by `sig3` at logic-1 at the next clock cycle. This is due to the clock cycle delay operator `##n` between signals `sig1` and `sig3` (with `n`=1). Once the antecedent is matched, the consequent must be matched starting in the next clock cycle. The consequent specifies that the signal `sig2` is to be at logic-1, then followed by signal `sig5`. For this assertion to pass, the circuit must adhere to the sequence of events defined in the consequent anytime the antecedent is observed. Since the assertion has explicitly defined the sequence of events that must take place, it then becomes possible to generate tests based on those events.

2.1.3 Test Generation Using NFA Representation of Assertions

Generating tests by using assertions takes advantage of the Boolean events and sequences explicitly defined inside the assertion itself. This thesis relies on a computable representation for generating tests, known as the Non-deterministic Finite Automata. To generate this representation, a hardware assertion checker tool is used, namely *MBAC*.

Boulé et al [6] have created a tool called *MBAC*, that generates hardware checkers from properties written in SVA or PSL under the simple subset guidelines. *MBAC* first generates a finite automaton representation of each property either in acceptance or failure mode. Following this process, a Verilog representation of the checker circuit is produced, which can then be used as a run-time assertion checker. The checkers are suitable for use in simulators and formal verification engines that do not support assertions; they can be instrumented as part of the design for post silicon validation, and they can also be synthesized in hardware for advanced debugging during the prototyping stages.

Figure 2.3 shows a high-level view of the assertion-based verification methodology, and the roles played by the assertions, the checkers and the checker generator. At the left of the figure are the given inputs to the tool, namely the Design Under Verification (DUV) and the assertions. In this illustration a failure-mode checker is pictured, along with its interconnection to the DUV for finding errors. Finally, the

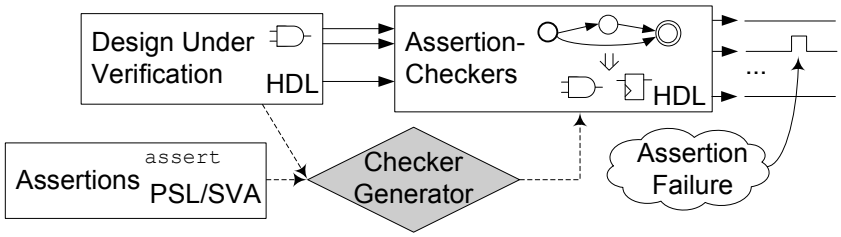


Figure 2.3: MBAC Checker Generator for Hardware Verification.

circuit-level checkers are derived from non-deterministic finite automata which is used as a *computable representation* for generating tests from assertions.

A *Non-deterministic Finite Automaton* (NFA) is a directed graph defined as a 5-tuple $\mathscr{A} = (Q, \Sigma, \delta, q_0, F)$, which is used for representating assertions. The non-empty set Q defines a set of states (nodes) , where q_0 is the initial state ($q_0 \in Q$) and F represents the non-empty set of final states , with $F \subseteq Q$. The set of symbols Σ represents Boolean expressions that reference either the primary inputs, primary outputs or intermediate signals of the DUT. A transition relation, represented by the set $\delta = \{Q \times \Sigma \times Q\}$, defines the required conditions for the automaton to activate its next state(s). It consists of a set of ordered triples $(s,\ \psi,\ d) \mid s, d \in Q$ and $\psi \in \Sigma$, where s and d represent the *source* and *destination* states respectively, and ψ represents a given *transition clause*, defined next.

Definition 3 (Transition Clause) *A transition clause ψ references the signals defined in Σ which explicitly describes the required conditions that must be satisfied in order for $\mathscr{A}$ to enter into its next state (or set of active states).*

Because the automaton is inherently non-deterministic, and since the assertion is clocked, the automaton can potentially transition to a new set of active states for each clock cycle.

The NFA representation for assertion ϕ_1 is depicted in Figure 2.4. The left part of the figure represents the *acceptance automaton* while the *failure automaton* is shown on the right. Both types were generated using the *MBAC* tool. Describing the steps of converting an assertion to its non-deterministic finite automata representation is beyond the scope of this thesis; however, the details are thoroughly described in [6]. An *acceptance automaton* describes the required sequence of Boolean events that creates a successful completion (passing) of an assertion. In contrast, the *failure automaton* shown on the right depicts the sequences that can lead to the failure of the assertion. Each edge is labelled with a *transition clause* ψ which references the signals that are defined in assertion ϕ_1 introduced previously. The initial state, $q_0 = S0$, is

(a) Acceptance Automata *(b) Failure Automata*

Figure 2.4: Non-deterministic Finite Automata Representation of ϕ_1

represented by a gray node while the final state is shown in green for the acceptance automaton ($F = \{S4\}$) and in red for the failure automaton.

Whenever an automaton has reached its final state, it is said the assertion has passed (acceptance automaton) or failed (failure automaton). In our usage, the acceptance and failure automata will serve as the *blueprint* for generating the test sequences that create either a passing or a failing of an assertion, respectively. Test sets that don't exercise the assertions have limited value in verification.

The NFA representations of assertion ϕ_1 is seen to have more than one path that leads to the final state. These different paths are referred as *test paths* of the assertion; the set of all test paths is denoted Π.

Definition 4 (Test Path) *Let π_i be a test path inside the NFA representation of an assertion that begins in the initial state, q_0, and ends in a given final state, q_f. A transition clause ψ references the signals that were defined in Σ, thus each test path π_i contains a set of transition clauses, Ψ_{π_i}, which represents the required ordering of transition clauses $\{\psi_1,\psi_2,...,\psi_n\}$ for passing (or falsifying) an assertion ϕ_i.*

For the cases used in this thesis, the Boolean signals defined in the assertions refer to the primary inputs of the DUT is assumed. When this is not the case, primary input justification is handled via model checking in order to obtain usable test sequences, given that internal signals and primary outputs can not be driven directly by the testbench. Thus, each test path π becomes a test sequence when assigning the appropriate Boolean logic values to each signal in the transition clause, ψ, which can then be directly applied to the DUT's primary inputs. For instance, from the NFA representation depicted in Figure 2.4, test path π_1 contains a set of transition clauses $\Psi_{\pi_1} = \{$`sig1`, `sig3`, `sig2`, `sig4`$\}$; these values thus become the test sequence used for verifying the circuit.

The background presented in this section will now be applied to the proposed test generation and compaction strategy, which are presented in Chapters 3 and 4 respectively. To assess the test quality from assertions, the technique of mutation testing is used, where the discussion is presented in the next section.

2.2 Assessing Assertion Quality with Mutation Testing

Assertion-based verification is playing a prominent role for validating the functional correctness of digital designs. Assertions can come from various sources such as from analyzing the design specification, to interacting with the designers where both can describe the design intent. A problem that is often faced by verification engineers is to determine the completeness of the assertion set. The lack of completeness can cause functional bugs to escape during verification. This can be due to an assertion that was incapable of capturing the error or that was not fully defined from the specification. Thus, it is imperative that the set of assertions is sound and of capable detecting potential design errors.

Assessing the completeness of an assertion set is becoming an important problem within the verification community. One of the methods in seeking the quality of assertions is to determine whether the design was able to exert all the necessary conditions in passing the assertions; however, this does not imply that the design is entirely bug-free because assertions can pass without being evaluated (known as a vacuous pass. This will be further explained in Section 3.3). Another method, that is effective, is to use mutation testing, where this method determines if the assertions are able to detect the functional fault inside a mutated design.

Mutation testing is a technique that is based on software testing, which is used for gauging the adequacy of tests [16]. The intent is to measure the capability of a test set for detecting mutations, usually in the form of functional faults, that are injected into the design. This technique relies on generating multiple copies of the Design Under Verification (DUV) where each instance contains an injected functional fault, known as a *mutant*. The injected fault can be as simple as a syntactical change in the design's Register Transfer Level (RTL) code. The purpose is to determine if a set of tests is able to *excite* (activate) the mutant, and propagate the erroneous result to the design's primary outputs. When the output of the mutated design differs to the reference design (mutant-free), then the mutant is deemed *killed* (detected). Otherwise, additional tests are required, which results in a strengthened test suite that is capable in detecting more faults.

Evaluating the completeness of an assertion set (A) is of paramount of importance in design verification. The lack of thoroughness in the set of assertions can potentially have design flaws go undetected and escape during the verification stage of the design. To evaluate the quality of assertions, the concept of mutation testing is used where the

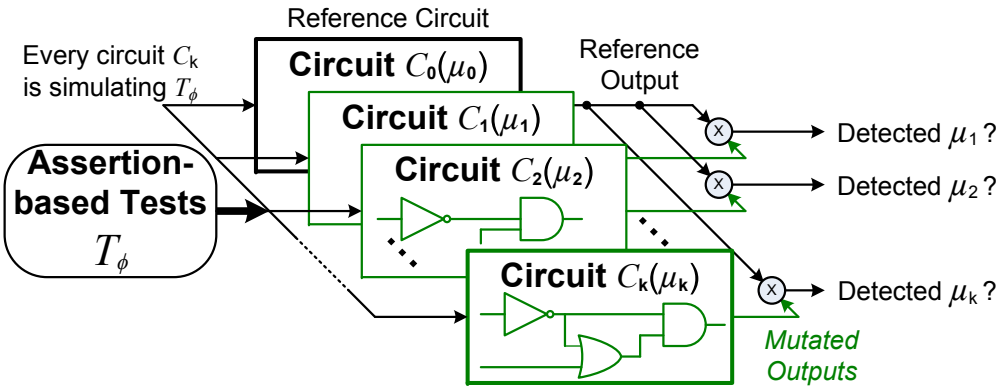


Figure 2.5: Mutation Testing using Assertion-based Tests

original circuit is injected with a set of functional faults, F. The dynamic verification environment is chosen because simulation is the most predominant method in industry due to its ability to scale with larger designs. A copy of the mutated design is created for every injected functional fault $\mu_k \in F$, as shown in Figure 2.5. It is important to note that μ_0 represents a *null mutant*, which does not affect the functionality of the circuit. The set of assertion-based tests T_ϕ, that was generated from a set of assertions A, is used for simulating the set of mutated designs. A mutant-free design containing the null mutant, denoted as C_0, is used for computing the correct result without having any injected faults. At the end of simulation of each mutated design C_k, the values of the primary outputs (O_k) are compared against the reference design (O_0). If any output from does not much with the reference design, then the mutant μ_k is deemed *killed*. Otherwise, is said to have *survived*.

After all the mutated designs have been simulated, an adequacy score is computed, known as the *mutation coverage*. Mutation coverage is defined as:

$$cov_\mu = \frac{|F_{kill}|}{|F|} \times 100\% \tag{2.1}$$

which is the ratio of the number of *killed* mutants ($|F_{kill}|$) to the total number of mutations ($|F|$, excluding the null mutant μ_0). The value of cov_μ gives a sense in how well the tests from T_ϕ were able to detect the injected mutations (i.e., the test quality). The goal of mutation testing is to raise the mutation coverage close to 100%. To improve the assertion-based test set for killing the remaining set of mutations, $\bar{F}_{kill}$, the verification engineer will derive additional assertions that will generate additional tests that can be added to the existing set, T_ϕ.

Mutation testing is an effective method for assessing the quality of the test set; however, it comes with high computational costs when injecting a large set of muta-

tions into the design. There are existing techniques for pruning the number of mutations to inject. *Selective mutation* minimizes the number of mutants by injecting a subset of mutations into the design without having any loss in the test effectiveness [25]. It has been experimentally proven that the number of mutations has been reduced to five key mutations, which helps in reducing the computational costs [26]; however, as designs become larger, using mutation testing during simulation becomes a performance bottle neck.

Graphics Processing Units are an ideal platform for accelerating scientific computations, especially in the field of Electronic Design Automation (EDA). Mutation testing can benefit from the use of GPUs. As seen in Figure 2.5, there is a certain amount of data parallelism that can be exploited. The next section presents an overview of GPU architecture and the OpenCL run-time execution model.

2.3 Accelerating EDA Algorithms on GPUs

Modern Graphics Processing Units (GPUs) are becoming easily programmable for performing not only graphics rendering, but also used in scientific applications. As Advanced Programming Interfaces (APIs) for these coprocessors have improved, many different forms of computation are being offloaded from the host CPU and onto GPU, with significant speed-ups being achieved [27]. Such popular APIs are the Compute Unified Device Architecture C (CUDA C) Programming language provided by NVIDIA [28] and the Open Computing Language (OpenCL) supported by AMD [29] and NVIDIA [30]. These APIs allow developers to harness the compute power of GPUs for accelerating compute intensive applications. Thus, coining the term *General Purpose-computing on Graphics Processing Units* (GPGPU).

Recently, GPUs have been used for accelerating compute intensive applications in the field of Electronic Design Automation (EDA) [31]. The massive parallelism provided by GPUs can help in accelerating circuit simulation algorithms; however, EDA algorithms generally involve several data dependent operations, which can pose significant challenges for efficiently parallelizing these algorithms using GPUs. Concepts in parallelizing and accelerating EDA algorithms using conventional processors have been borrowed and applied to GPU architectures. Previous researchers have thoroughly investigated in accelerating circuit simulation algorithms over a network of workstations, with the intent in reducing the simulation time [32]. These approaches are broadly classified into three different classes, namely *algorithm-parallel*, *model-parallel* and *data-parallel* [33]. Algorithm-parallel approaches assign different

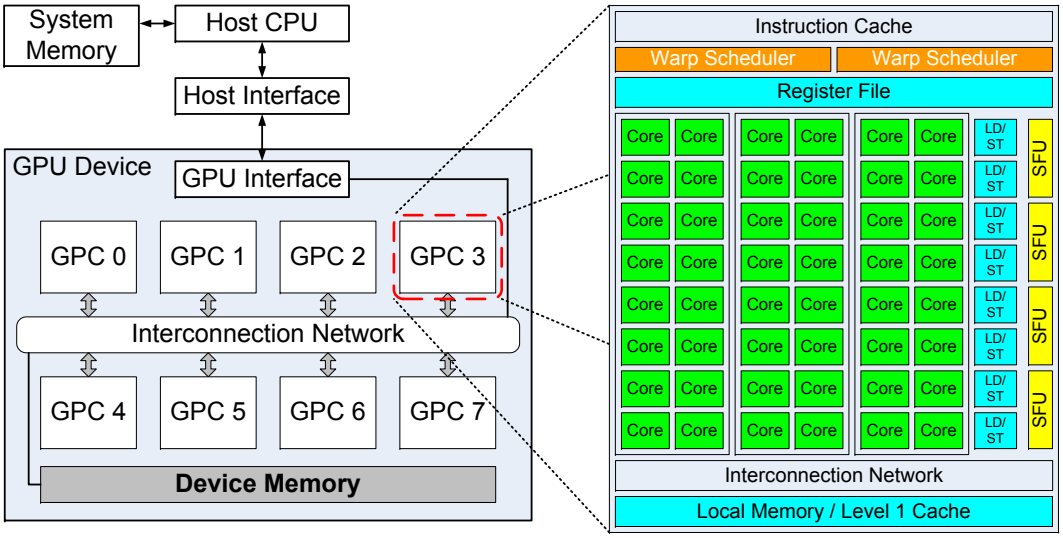


Figure 2.6: GPU Architecture

tasks within the simulation algorithm, such as scheduling and gate evaluation, and are distributed over different processors, working in a pipelined fashioned [34, 35]. Model-parallel involves with partitioning the circuit into disjoint sub-circuits, where each circuit can be logic (or fault) simulated independently with different processors [36]. Data-parallel algorithms can be further decomposed into *pattern-* and *fault-*parallel approaches. In pattern-parallel [37], all the test sequences are simulated concurrently, whereas in fault-parallel approaches simulate every fault independently [38].

The data-parallel approaches are suitable for the GPU, as they do not require inter-communication between other processing cores. The remaining challenge is to generate an efficient data-parallel representation, which ensures all the threads on the GPU operate on the plethora of circuit, fault and test data independently. The next section presents an overview of the GPU architecture and the OpenCL Execution Model.

2.3.1 GPU Architecture and OpenCL Execution Model

GPUs employ a single instruction and multiple data paradigm so that many instances of a kernel function can be executed concurrently. Figure 2.6(a) shows the architecture of a Fermi-based NVIDIA GPU connected with the Host CPU through the Host Interface. The architecture contains 8 Graphics Processing Clusters (GPC), each consisting of 48 processing cores, as shown in Figure 2.6(b) [39]. When a kernel

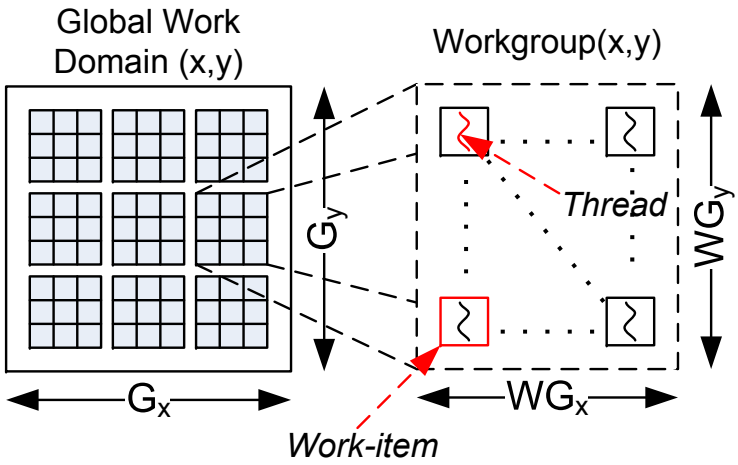


Figure 2.7: OpenCL Execution Run-Time Model

function is launched on the GPU hardware, each GPC will contain an instance of the kernel code. The individual processor cores (shown in the green boxes) execute multiple copies of the kernel code and operates on different data that is stored in the GPU's device memory. It is important to note that each processor core executes the same set of instructions in a lockstep manner.

Synchronization and data sharing within the GPC takes place by accessing the available local memory. Each instance of the kernel function has its own private memory that is accessible through the register file. Global memory can be read and written by all the processing cores on the GPU device; however, every access to device memory creates a latency of 400 to 800 clock cycles.

Figure 2.7 shows the execution run-time model of the OpenCL programming language. The developed simulation kernels that will be presented in this thesis are written in the OpenCL language, which gives the opportunity in running the kernels on other GPU architectures. The constructs and the execution model are similar to the CUDA C programming language [30]. In OpenCL, each kernel instance is known as a *work-item* and groups of work-items are known as *work-groups* [40], as depicted in Figure 2.7. The combination of all the work-groups define what is called a *global work domain*, which specifies the number of work-items that are executing on the GPU. Each instance of a work-item has its unique global ID that is accessible by the built in kernel function, `get_global_id()`, while the size of the global work domain is accessible through the kernel function, `get_global_size()`. When a work-group is assigned to execute on a GPC, the thread hardware schedulers decompose the work-group into 32 work-items, known as a *warp*. To achieve maximal performance, every work-item within a warp must have a common execution path.

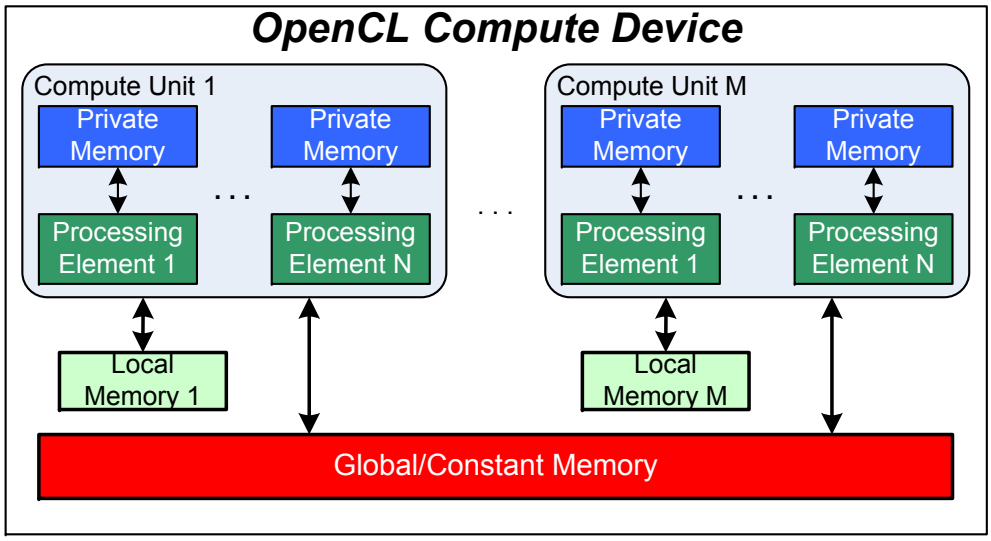


Figure 2.8: OpenCL Memory Hierarchy

2.3.2 Memory Hierarchy

Figure 2.8 shows the OpenCL memory hierarchy to which every OpenCL application must abide [40]. It is important to note that each type of memory in the hierarchy has different access latencies. The OpenCL *Compute Device* is represented by the GPU, which it contains a set of *Compute Units*. These depict the GPCs of the GPU. Each compute unit contains a set of *processing elements*, which represent the processing cores in every GPC. Every processing element from different compute units can access the *Global / Constant Memory* of the compute device. Every compute unit has a dedicated, high bandwidth local memory, that every processing element within a compute unit can access for sharing data or synchronizing work-items within a work-group. The processing elements have dedicated *private memory*, which represents the register file of each GPC. Work-items can access the register file for storing temporary data; however, the data stored in private memory cannot be shared with other work-items. The kernel code will use the built-in OpenCL kernel functions (e.g., `get_global_id()`) for retrieving the appropriate data element that is stored either in global, local or private memories.

Programming an application on the GPU is a non-trivial task, and many key factors must be exploited in order for the application to achieve optimal performance [41]. One of the factors is to have every work-item within a warp execute the same set of instructions that follow the same execution path. Any work-item within a warp executing different instructions leads to *branch divergence*, which causes the

hardware schedulers to execute each branch path in a serial manner, thus causing a decrease in performance. Accesses to the GPU's device memory must be performed in a coalesced manner in order to maximize the throughput. This requires all the work-items to access the device memory using contiguous index address values so that multiple memory requests can be fulfilled by a single transaction. Lastly, every GPC is equipped with a certain amount of local memory that is accessible by all work-items within the work-group. Storing most frequently accessed data into local memory can increase performance since these data are accessible in one clock cycle, thus reducing accesses to the GPU's device memory.

These factors have been exploited in the proposed simulator tools, namely *μ-GSIM* and *GS-SIM* that are presented in Chapter 5. The tools that were developed use a novel data-parallel generation and encoding scheme for efficiently mapping a plethora of circuit and mutant data onto the GPU's device memory. Additionally, a mutation testing environment for assessing assertion quality was also developed, which uses the same data-parallel representation and is presented in Chapter 6. The developed GPU applications have shown significant performance improvements, which is much needed in the quest for improving assertion and test quality.

2.4 Summary of Related Work

This section presents a discussion on the previous research work related to assertion-based test generation, test compaction, GPU logic and fault simulations and applying mutation analysis on assertions.

2.4.1 Related Work on Test Generation from Properties

A large amount of work on generating test sequences from properties originates in the field of software testing [42]. Closest to this work, *Oddos et al.* [2] recently developed a tool (*MyGen*) to obtain test vector circuit generators based on automata created by *MBAC*. Their approach relies on producing the test sequences pseudo-randomly with a Linear Feedback Shift Register (LFSR) and Cellular Automata. The test sequences satisfy a given property based on the *acceptance automaton* of that property. The hardware generators were synthesized for FPGAs; however the size and complexity depends entirely on the transition conditions (Boolean expressions) of each edge.

Koo et al. [43] proposed a bounded model-checking method for validating pipelined processors. They generated a model of the MIPS processor in graph form, where a

node represents either a pipeline (fetch, decode, execute, writeback) or a storage unit (memory or registers), while an edge depicts a data storage link. The objective is to generate a counter-example program that violates the user defined property. This goal is achieved by traversing the model (graph) of the microprocessor by finding a set of instructions that causes the property to fail. Similarly, *Mathaikutty et al.* used model checking for generating directed tests for designs written in SystemC combined with a set of specifications.

Shimizu et al. [44] presented a method for generating test sequences based on constraints written as Boolean formulas. The constraints rely on the state variables from previous states that are used for biasing during subsequent test generations. These constraints are then converted into a Binary Decision Diagram (BDD) which is used for generating test sequences that lead to a "true" value. These sequences are the inputs to the circuit that will exert the expected (good) behaviour of the design. This state space search is repeated for every dynamically created BDD at each clock cycle.

Calamé et al. [45] demonstrate automata techniques for finding test sequences that lead to a property failure. This method requires the specification of the design and the test purpose. The specification is described in the form of a finite state machine, while the test purpose is described by an automaton representing the constraints that guide the generation of test sequences. Performing the product of the specification and the test purpose automata creates the search space for test generation. State space traversal algorithms are used to create the test sequences for property failure.

Cheng et al. [46] generate tests sequences from an Extended Finite State Machine (EFSM) representation of a DUT expressed in VHDL or C. Their method exhaustively traverses all edges of the EFSM, however, the EFSM does not incorporate nondeterministic behaviour, whereas an automaton generated by *MBAC* does.

Test sequences from assertions can also be derived by traversing through an extended finite state machine representation of a design in formulating a set of constraints. A model checker will then be used as a constraint solver in which the results produced from the tool become the test sequences that satisfy the set of assertions [47]. Classical Automatic Test Pattern Generation (ATPG) algorithms have been employed with model checking where the assertion is represented as a stuck-at-0 or stuck-at-1 fault. A SAT engine attempts to solve a set of Conjunctive Normal Form (CNF) clauses representing the DUT for the attempt of detecting those faults that represents assertions [48].

Another approach uses logic implications [49] and well established implication-

based techniques [50]. In this method, logic implications define the expected relationships between values at different circuit sites. This technique helps in deriving useful tests that exercise the proper behaviour defined in the logic implications.

Assertions can also be used in guiding the test generator to produce test sequences without the presence of the DUT. *Pal et al.* [22] proposed a "black-box" approach for performing test generation from user-defined assertions. The black-box setup is somewhat interesting to us because it is aimed at cases when the DUT structure is not available, and creating the tests should proceed without the DUT being considered, such that the algorithms are independent from the size and implementation of the DUT. In this approach, assertions are seen as coverage points and are used as a source for test generation. The test set produced is applied at the primary inputs so that the DUT is able to reach those coverage points by exerting the specified events that are defined in the set of assertions [22, 7].

2.4.2 Related work in Test Compaction from Properties

Considerable research efforts have been made on developing novel techniques for reducing the size of a test set. The idea is to identify redundant test vectors that can be modified or removed from the set while maintaining the same coverage goals. Test compaction algorithms are broadly classified into *static* and *dynamic* techniques. Static compaction is applied after the test generation process is complete [51]. Although simpler to integrate as a post-processing step to an existing methodology, the full size of the test set must still be generated and handled. Test compaction algorithms that are directly integrated into the test generator are dynamic test compaction techniques [52]. They require modifications to the generator to allow modifying or removing redundant tests during the generation stage.

Techniques were proposed that dynamically or statically compact tests for detecting manufacturing faults of scan-based designs. With such techniques, the circuit's flip-flops are directly initialized through a dedicated port in order to bring the circuit into a particular state prior to applying the tests at the primary inputs. Fault simulators are typically used for dynamically compacting tests in which they determine the number of faults that were detected by a single test. The test itself can be modified either using genetic algorithms [53], complementing bits of a test vector [54] or deterministically assigning logic values to the *don't care* bits of a test vector [55] so that more faults can be detected by a single test. One of the proposed static test compaction techniques decomposes each test vector into its atomic components, then

identifies if each component detects the same fault [56]. Changing the sequence of test vectors by means of test vector reordering can potentially increase the number of detected faults and further reduce the size of the test set [57]. Set covering [58] and integer linear programming [59] have also been explored for compacting tests statically.

Test sizes for detecting manufacturing faults of non-scan circuits is larger than scan-based designs because additional test data is needed for performing the scan-in and scan-out operations at the circuit's inputs. Several techniques have been proposed for reducing test data by minimizing the number of scan operations in between tests that are applied at speed. One of them is to merge at-speed tests if they do not require scan out operations for detecting the faults [60], then use scan-based circuit test compaction techniques [61] to further reduce the test set. Compacting tests with partial scan techniques has been explored in [62] where a subset of flip-flops are used for shifting in and shifting out the stored values that determine the detectability of faults. Further test reduction was achieved by configuring the flip-flops in the circuit into a shift register for shifting the detected fault data and the elimination of the scan inputs and outputs [63]. This in turn reduces the amount of test data when scan operations are not needed.

Assertion (property) clustering methods have been explored for various purposes. The key idea is to group similar assertions with a certain level of similarity that can be beneficial to other assertions for a specific application. For instance, grouping similar properties together can help in generating tests for multiple assertions and can accelerate the test generation process so that the similarities between properties can be reused [64]. Similarly, assertion clustering can also be applied for generating an efficient cluster of hardware assertions that produces minimal hardware usage and reduced power consumption that is suitable for many post-silicon debugging infrastructures [65] [1].

Assertions can also be clustered on a reprogrammable fabric such as an FPGA. Clusters of similar assertions were formed in order to minimize the number of reconfigurations needed, which is suitable for debugging designs using rapid prototyping methods that require reprogramming the FPGA many times [66]. This concept was also explored for Time Multiplexed Assertion Checkers where a group of similar assertions are confined to a portion of the FPGA [67].

Little research has been made on compacting test sequences from assertions when

[1]The authors used clustering of hardware assertion checkers whereas in this work, we use clustering of assertions themselves

using clustering methods. One proposed approach is to compact directed tests from properties for validating a pipeline processor. This is achieved by analyzing the finite state machine model of the processor by identifying and removing redundant and unreachable states along with illegal transitions in order to minimize the overall state space of the model [68].

2.4.3 Related Work in GPU-based Logic and Fault Simulation

GPU-based logic and fault simulation and its applications have been extensively studied by various authors. In each implementation, every author has explored different methods for mapping the gate and fault data onto the GPU's device memory for exploiting the inherent parallelism offered by GPUs. In the work of *Li et al.* [69], the authors map the circuit's data using the GPU's computer word, where the different threads decode the gate and fault data information. Each gate containing the faults was encoded using a 32-bit word, while also storing the input port numbers for identifying the location of the faults. *Koshte et al.* [70] proposed a Parallel Pattern Single Fault Propagation algorithm on the GPU where they encoded the different gate and fault data over the computer's word. They limited the number of inputs of every logic gate to two input ports, so they can constrain the memory size of the circuit and fault data. Accelerating fault simulation algorithms was also performed by *Gulati et al.* [33]. Their approach was to map the gate data using specific integer identifiers, so that each thread computes the effective address in order to obtain the appropriate logic value by means of a truth table stored in shared memory. The fault data was encoded over the computer word by storing integer values indicating the port containing a particular stuck-at fault.

In the context of logic simulation, *Chatterjee et al.* [71] proposed an event-driven logic simulator based on a look-up table method for obtaining the correct logic value. Additionally, they have partitioned the circuit with their proposed cone clustering algorithm so they can generate a set of macro-gates for simulation. *Sen et al.* [72] also proposed a logic simulator by representing the circuit as an And-Inverter Graph. They have also used a form of partitioning for isolating a set of gates that are responsible for evaluating a particular set of primary outputs [73]. In the work of *Bombieri et al.* [74], the authors developed a logic simulator that is based on designs written in SystemC. They transformed a SystemC design into a C++ representation that is executed on the GPU. Additionally, they performed a thorough study by comparing the run-time

performances between OpenCL and CUDA on the NVIDIA GPU platform [75].

GPU-based logic and fault simulation algorithms have also been applied in other VLSI related applications. *Holst et al.* developed a scan-based power simulator that is based on a form of logic simulation that is implemented on GPUs [76]. Their approach was to simulate multiple instances of the circuit using different input waveform data for computing the power dissipation for every logic gate. *Li et al.* also proposed a GPU framework for generating tests in design verification [77], in which a GPU-based logic simulation algorithm was used for justifying and assigning logic values to the internal signals within a circuit. These tests are ultimately used for verifying the correctness of the design. Additionally, they also proposed an algorithm for reliability analysis of logic circuits using GPUs in which they also use a form of logic simulation [78]. Fault table generation also uses a form of fault simulation algorithms on GPUs and was studied by *Croix et al.* [31].

2.4.4 Related Work on Mutation testing with Assertions

Mutation testing is an approach that is based on software testing which is used for gauging the quality tests [79]. This approach injects intended faults, called *mutants*, into the original program one at a time and determines if the set of tests are able to excite (*mutant activation*) and detect (*killed mutant*) the fault that was introduced. The primary objective is to determine if additional tests are needed so that the quality of the test suite can be improved. Recently, concepts of mutation testing were borrowed and applied to the functional validation of digital circuits. The authors *Sousa et al.* [80] injected different mutation operators into the SystemC model whereas *Bombieri et al.* [81] applied mutations into the extended finite state machine representations of the Transaction Level Models (TLM) of a design. These approaches were used for assessing the quality of tests generated from a testbench that is used for validating TLM designs written in SystemC.

Behnam et al. [82] also used mutation testing techniques where they replace a set of gates with a different logic type representing mutations of a synthesized gate-level circuit. They transformed the mutated circuit into a set of Conjunctive Normal Form (CNF) clauses and perform satisfiability in order to determine if the mutants were detected. Similarly, *Mirzaeian et al.* [83] also used the idea of gate replacements by injecting erroneous multiplexers into the design, which then gets converted into CNF clauses for word-level satisfiability solving. Results from the SAT solver determine if the faulty multiplexers were detected and indicate the location in the RTL code that

contains the intended errors. These gate replacement techniques are also used in our work as mutations for synthesized gate-level circuits and are further explained in [84].

Mutation testing has also been applied for assessing the quality of properties in validating the correctness of a design. The objective is to measure the *property coverage* which gauges how well the specified properties can detect the set of injected mutants inside the original design [85]. Previous research work, quantifying property coverage was also used in formal and dynamic verification methods. *Kupferman et al.* [86] proposes a formal verification method by employing a model checker for analyzing the completeness of properties. Their approach perturbs the original finite state machine model with a set of mutations (faults). The objective is to analyze the properties' thoroughness so that each property can be satisfied (or falsfied) non-vacuously, ultimately improving the quality of them. Similar work was also proposed by *Fraser et al.* [87] in which they use model checking for generating counterexamples when properties detect the injected fault. The goal of their work is to improve the test quality by adding properties that can detect all of the injected mutations. Similar formal verification techniques have employed by various authors where they use mutation testing for analyzing a set of properties that can be easily satisfied without thoroughly exercising the specified sequence of events [88, 89], also known as vacuity analysis [90].

Dynamic (simulation) approaches have also been proposed for analyzing the thoroughness of written properties. Simulation is still the predominate method in industry for design validation [91]. *DiGuglielmo et al.* proposes a method for analyzing vacuity analysis of properties using fault simulation [92]. Their approach is to inject faults at the RTL level of the design and map them using a gate-level stuck-at fault model. Then, the fault simulator will determine which fault was detected, which determines which property was able to detect the fault. Similarly, they also used this method for quantifying property coverage by means of vacuity analysis, completeness and overspecification [93].

Banerjee et al. [94] have also used a form of simulation for assessing how well the assertions were able to uncover the injected stuck-at faults at the interface of the design. Similarly, *Pal et al.* [22] developed a testbench in the attempt of accelerating assertion coverage in terms of property completeness. They treat assertions as coverage points inside the design. Their goal is to have the testbench generate the required stimuli so the circuit is able to reach those coverage points, thereby measuring the property coverage. Other research work undertaken by *Di Guglielmo et al.* [95] is closely related to ours where they accelerated simulations by developing a parallel

functional simulator. Their approach is to synthesize every fault injected circuit and to rely on the bits of the computer word for detecting the faults. The input stimuli they used are based on the properties that were associated to each design. They subsequently used their developed simulator for quantifying the property coverage.

2.5 Chronology Work Overview

This section gives an explanation of how the previous work relates to the development of an assertion-based test generator and GPU accelerated mutation testing simulator, with the ultimate goal for providing an infrastructure for assessing assertion quality.

2.5.1 Coverage Driven Assertion-based Test Generation

Chapter 3 discusses our developed assertion-based test generator, *Airwolf-TG*. The developed tool is capable of generating tests by using a *computable representation* of assertions, namely NFA, which was described from Section 2.1.3. A similar tool developed by *Y. Oddos et al* [2], called *MyGen*, also uses the NFA representation of assertions for generating assertion-based tests; however, their approach was to pseudorandomly exert different transition clauses that creates an assertion pass. Our approach that was incorporated into *Airwolf-TG*, employs a directed strategy for generating assertion-based tests by thoroughly exploring the NFA space, thereby all of the transition clauses are exerted at least once. A set of proposed relations between automata coverage and assertion coverage metrics (explained in detail in Section 3.3), is incorporated into the developed test generator which helps in generating directed tests from assertions *non-vacuously*. These coverage metrics have shown that the generated assertion-based tests from *Airwolf-TG* attained improved coverage when compared to the tests generated by *MyGen*.

2.5.2 Test Compaction Techniques for Assertion-based Test Generation

Chapter 4 describes two novel test compaction algorithms, which were incorporated into the developed tool, *Airwolf-CTG*. The proposed test compaction algorithms uses assertion clustering by grouping assertions containing similar signals and sequences. This work was closely related to the ones found in [65, 66, 64], where authors have used clustering for the context of generating an efficient cluster of hardware assertions

for post-silicon debug, reducing area usage on a programmable fabric such as FPGAs, and also for accelerating the test generation process of assertions by exploiting similarities. The test compaction techniques that were incorporated into *Airwolf-CTG* also uses assertion clustering. Our approach exploits the similarities that are found within a cluster of similar assertions, which can then be shared for compacting the assertion-based tests. This has led to the development of the *Test Path Overlapping* compaction algorithm, described in Section 4.4.1. An improved test compaction algorithm, namely *Parallel Path Removal* (explained in Section 4.4.3), leverages the unspecified conditions of assertions for compacting tests within a cluster. This has helped in providing additional compaction of the assertion-based tests that is beneficial in reducing the overall applied verification time.

2.5.3 Efficient Data Encoding of Mutation and Fault Data on GPUs

Chapter 5 presents our proposed efficient data-parallel representation and encoding techniques for accelerating mutation and fault simulations using GPUs. As discussed from Section 2.3, one of the challenges for using GPUs for accelerating EDA algorithms is to ensure every thread (work-item) is operating on independent data as much as possible. Previous approaches that were found in [33, 71] performs logic and fault simulation by having every thread on the GPU to compute the Boolean logic of every gate using a look-up-table method. The proposed approach presented in this chapter transforms the circuit into a stream-based representation, so that every thread operates on a single instance of a circuit. Additionally, we also show how the GPU's computer word is leveraged for efficiently encoding the different gate types, mutant and fault data, which has led to the two encoding techniques, namely Multiple Mutant Gate (used by *μ-GSIM*) and Multiple Fault Gate (used by *GS-SIM*). The purpose of these encoding strategies and a stream-based representation of the circuit is to reduce the overall memory usage on the GPU's device memory. This has enabled for more threads to simulate multiple circuit streams while operating on independent mutants or faults, thus improving simulation efficiency on the GPU.

2.5.4 Using GPUs for Accelerating Mutation Testing of Assertions

Chapter 6 shows our proposed framework for assessing assertion quality using mutation testing on GPUs. This chapter combines the test generation strategy that was described from Chapter 3 and the data encoding techniques that were shown in Chapter 5. Similar work in terms of assessing assertion quality was proposed by *Pal et al.* [22], where assertions are used as *coverage points* (discussed from Section 2.1.1) for assessing how thorough the design under verification was explored and are able to catch the injected faults [94]; however, their approach was to pseudorandomly generate tests and apply them at the inputs of the design while observing which assertion has passed and faults were detected. Our proposed approach uses the directed assertion-based tests from *Airwolf-TG* (described from Chapter 3), which directly exerts the necessary conditions for passing the assertion. Additionally, we inject synthesized mutations into the circuit and determine if the assertion-based tests were able to detect the mutant during simulation on the GPU.

Chapter 3

Coverage Driven Assertion-based Test Generation

This chapter presents the proposed assertion-based test generation tool *Airwolf-TG*: A Test Generator for Assertion-based Dynamic Verification. *Airwolf-TG* operates over NFA representations of assertions, which were discussed in Section 2.1.3. A set of coverage metrics for NFAs is also proposed, which helps gauging how well the assertion was evaluated during simulation, and also for generating tests non-vacuously. The proposed tool will help in performing an effective dynamic verification of digital designs.

3.1 Motivation

Dynamic (or simulation-based) verification is still the dominant approach for verifying the Design Under Test (DDT). It requires a DUT simulation over sufficiently detailed test sequences, to ensure conformance to expected behaviour. Most commonly, verification engineers are asked to perform functional verification, which typically amounts to exercising the expected *good behaviour* by generating test sequences that should produce the anticipated response from the circuit. To exercise the circuit more thoroughly, one should also attempt to find test sequences leading to faulty behaviour.

Assertions help by increasing the observability within the circuit, and can also help to create behavioural scenarios that can be seen as potential coverage criteria [4, 96]. Properties are commonly defined using modern assertion languages based on linear time temporal logic and sequential extended regular expressions. Additionally, they serve as a formal specification mechanism to define the intended behaviour in a design. Any deviation causes an assertion to fail, which can be captured by either

the simulation environment or by formal methods.

The work presented in this chapter addresses the question: *Have we produced enough test vectors to exercise most of the possible behaviours of the circuit?* We seek to obtain the minimal number of test vectors that thoroughly exercise the circuit behaviour based on the defined properties. It is assumed that enough assertions were written to properly specify the functionality of the circuit. Test generation is an essential step in the dynamic verification process, with the main goal being quantified in the form of *coverage* metrics.

Within ABV, we could actually use the readily available assertions to produce test sequences. If assertions thoroughly define the properties of the design, then they also provide a blueprint for exploring the relevant common cases as well as the corner-case scenarios. For that, it is imperative to have *sound assertion coverage metrics* that measure how fully the properties have been exercised [3]. Hence, the coverage goals can be thought of as the exploration of the specification space. This chapter proposes proposes a set of coverage metrics for automata that represent assertions, and relate them to the specific assertion coverage goals. The contributions that are made in this chapter are as follows:

- We present an approach for verification test generation based on properties represented by finite automata;
- We establish a relation between assertion coverage goals and the automata coverage;
- We develop a method for detecting and exploiting non-determinism for efficient automata traversal;
- We also develop a mechanism for analyzing coverage for a set of vectors that is used in determining the edge coverage and for combining vector generation schemes.

The contents of this chapter are organized as follows: [1] Section 3.2 shows our proposed methodology for generating tests from assertions. Section 3.3 presents a thorough discussion on vacuity in testing, assertion coverage, different automata coverage metrics and test coverage which are all valuable in test generation. Section 3.4 presents the underlying algorithms and Section 3.5 details the experiments undertaken with *Airwolf-TG*.

[1] The contents of this chapter is based on the article entitled *Defining and Providing Coverage for Assertion-based Dynamic Verification* [7] and the paper *Airwolf-TG: A Test Generator for Assertion-based Dynamic Verification* [8]

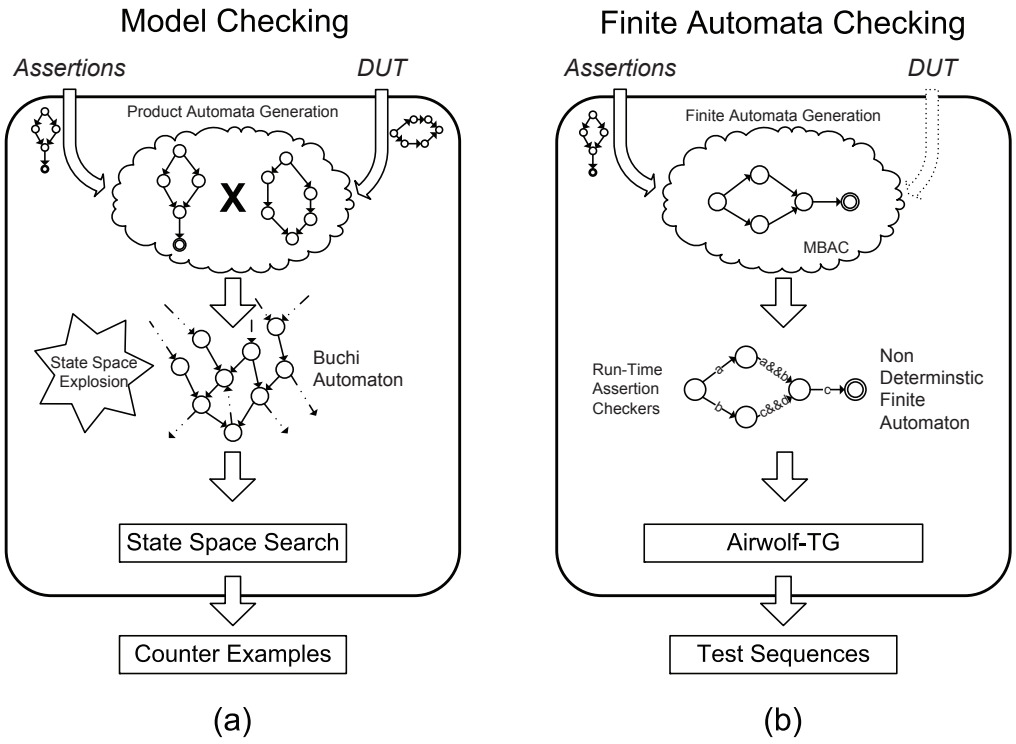


Figure 3.1: Model checker vs. finite automata checker in model-based test generation

3.2 Finite Automata Checking

Most commonly, generating test sequences in the model-based approach amounts to generating a witness trace or a counterexample in Model Checking (MC). MC uses a finite state machine description of the circuit and a given property that it must satisfy. From these two inputs, it generates a *product automaton* that describes acceptable behaviours for infinite input sequences. While the product automaton structure, typically a Büchi automaton, allows us (in principle) to verify whether the properties hold for infinite traces, this method often suffers from state space explosion and is typically prohibitive for realistic circuits.

The model checking approach is illustrated in Figure 3.1(a). Table 3.1 shows the major differences between the model-based approach using MC and our proposed Finite Automata checking approach. The key point to make is that the state explosion associated with MC-based test generation is avoided at this stage by relying *only* on the checker automata to devise the sequences of symbols exercising the assertion automata.

The finite automata that describe properties are generated by the *MBAC* tool [6],

which produces assertion checkers suitable for dynamic verification. The tool explicitly optimizes the Non-deterministic Finite Automata (NFA) for subsequent generation of assertion checker circuits. With this approach we utilize the single automata from properties (Figure 3.1 (b)), as opposed to the product automata.

Our test generation tool, *Airwolf-TG*, requires that the original assertion coverage goals be expressed in terms of automata coverage. When assertions reference internal signals, additional steps must be employed for the *Airwolf-TG* assertion sequences to find the corresponding primary input sequences of the DUT. We focus on the assertion automata alone and their test sequences, as well as the much needed coverage metrics related to assertions. Verification techniques ignoring the DUT in whole or in part are sometimes encountered (i.e. abstraction-based methods [97]), hence this work can be potentially useful from that angle as well.

3.3 Coverage in Assertion-Based Verification

To show the need of coverage in assertion-based verification, we initially describe the issue of vacuity for passing or failing assertions vacuously.

3.3.1 Vacuity in ABV

Vacuity is a common issue that needs to be addressed in the ABV paradigm [98]. Upon simulating a test sequence, an assertion should succeed by having the design perform the expected behaviour as specified in the property; however, it can also succeed if the design does not produce any signals exercising the property.

For example, the following SVA assertion illustrates a simple grant and request property:

```
assert property (@(posedge clk) req |-> ##1 gnt)
```

Table 3.1: Model Checking vs. Finite Automata (FA) Checking

Category	Model Checking	FA Checking
Automata Type	Büchi	NFA
Represented Traces	Infinite	Finite
Use of Product Automata	Yes	No
Computation Time	Potentially huge	Low
Output Type	Counter Examples	Test Sequences
Input HDL	Yes	No

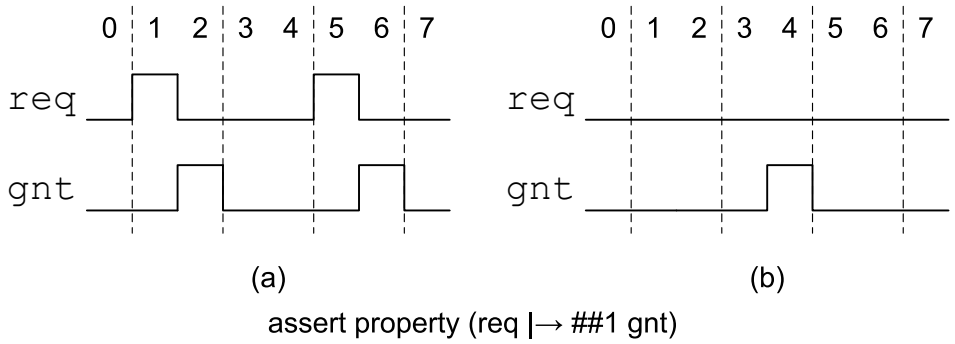


Figure 3.2: Waveform for SVA assertion

Figure 3.2 shows the two timing diagrams that can satisfy the property. Part (a) in the figure depicts that for every assertion of signal `req` at clock cycles 1 and 5, the signal `gnt` must be asserted on the next clock cycle (i.e. at clock cycles 2 and 6 respectively). On the other hand, as shown in Figure 3.2(b), if the `req` was not asserted throughout the simulation, then this property is satisfied *vacuously* without effectively exercising the antecedent portion of the assertion [99]. When this phenomenon occurs, there is a possibility that the design may not have produced the antecedent signal, the property itself was not thoroughly specified or the tests did not fully exert the specified signalling conditions in the assertion [5].

To achieve effective test generation from assertions we first need to derive suitable coverage metrics for generate tests that passes or fails the assertion non-vacuously. The intent is to map the high-level verification goals to the automata coverage goals, such that we only need to worry about the automata coverage during the test generation.

3.3.2 Assertion Coverage

Coverage is an important metric that measures how well an assertion captures the intended design behaviour. Since our goal is to fully exercise assertions by dynamic verification, we would naturally want to attain a good coverage of the assertion source code. In practice, one could consider the evaluation/firing of the assertion to constitute a coverage metric [100]. That by itself is not a sufficient form of coverage since there are different sequences of internal states that can satisfy or falsify the property.

As the attempts to define coverage in conjunction with assertions were limited, we first recall a few concepts from source code coverage metrics, such as: *statement (line), expression, branch and path coverages* [101, 16]. When narrowing the scope to

assertion code, difficulties arise as modern assertions languages allow a much higher density of code (often into single-line statements) such that line coverage might not produce sufficient granularity in assertion coverage. Similarly, branch and expression coverages from high-level languages and RTL do not have their meaningful equivalent in assertions.

A more effective assertion coverage strategy is proposed in [102] to help the verification engineer determine the following aspects about the defined assertions : Are they exercised thoroughly? Are their Boolean expressions thoroughly assigned? Regarding to the first question, the assertion statement should thoroughly exercise all the temporal steps in sequences; this is referred to as *Assertion Step Coverage*. For example, the property:

```
a |-> ##1 (b[*0:3] ##1 c)
```

is satisfied by requiring signals `a` and `c` to occur; however, signal `b` can remain low throughout the simulation.

The second question above concerns the Boolean variables referenced in a property. *Assertion Variable Coverage* determines which variables were assigned or remained dormant during simulation. For example, the property:

```
a |-> ##1 (e||f)
```

is satisfied by requiring signal `a` to be asserted and either `e` or `f` to occur. Any of the two Boolean variables in the `OR` expression can remain dormant throughout the simulation.

3.3.3 Mapping Assertion Coverage to Automata Coverage

In this section, we will briefly describe each automata coverage metric that will be applied to the assertion-checker automata produced by *MBAC*. From there we will try

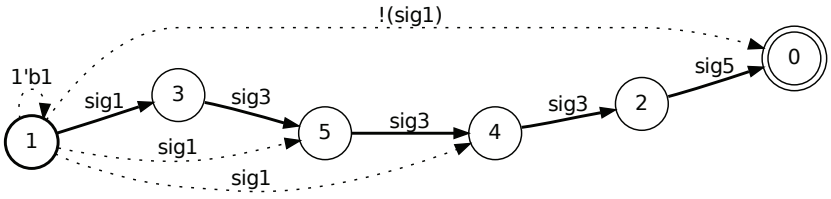


Figure 3.3: Node coverage metric

to map each coverage metric with the assertion source code coverage metrics briefly defined in the previous section.

Node Coverage expresses how many nodes of the entire graph have been visited, and is the most common metric. When covering all the nodes, the assertion (or the automaton) has entered into all of the possible states for that property. Shown in Figure 3.3 is an automaton traversed with node coverage as the goal. It is evident that in node coverage not all edges are traversed (dotted lines), which can result in test sequences that vacuously satisfy a property.

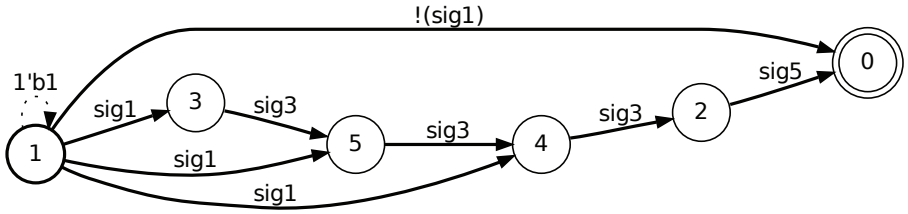


Figure 3.4: Edge coverage metric

Edge Coverage is one of the most widely accepted coverage metrics, in which the intent is to traverse all the edges of the automaton. When achieving the edge coverage, we ensure that all the Boolean expressions of the property are included during test generation at least once. Figure 3.4 depicts a graph with complete edge coverage. When performing this metric, it guarantees complete node coverage, and generates extra test sequences to guarantee that a given property is satisfied non-vacuously. A correlation between edge coverage and assertion source code coverage would be *Assertion Variable Coverage* as it requires all variables to be true or false at least once.

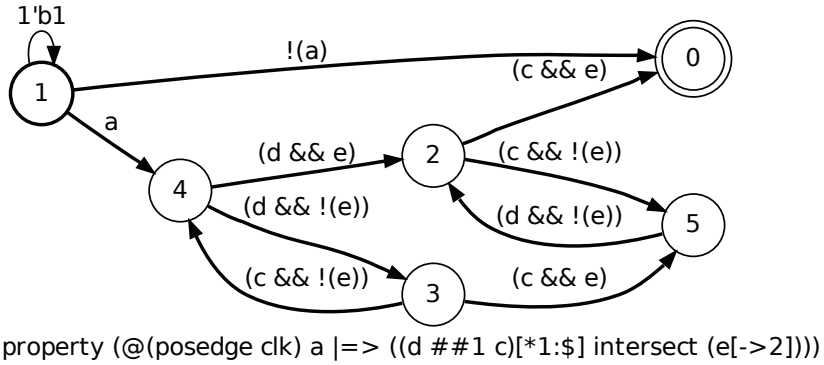


Figure 3.5: Complete round trip coverage metric

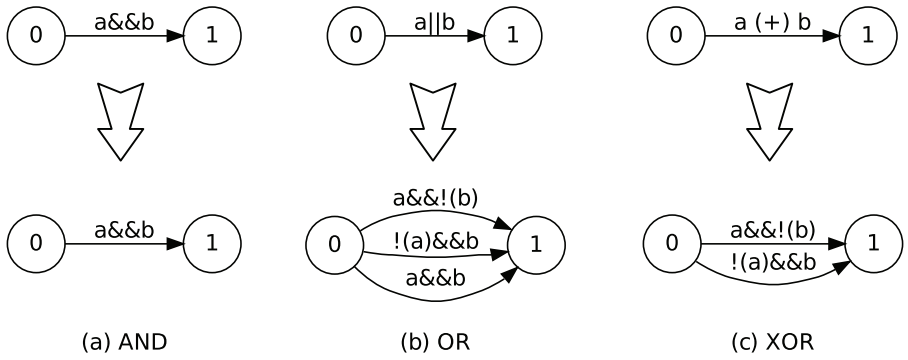


Figure 3.6: Edge completion for *and*, *or* and *xor* Boolean expression coverage

Complete Round-Trip Coverage (CRTC) involves covering all the round-trip paths (or cycles) that exist in the automaton. This type of coverage falls under *Expression* and *Path Coverage* in the assertion coverage criteria. At least one path must contain a cycle in order to cover all the edges of the graph as seen in Figure 3.5. It is apparent that the paths also attain complete node and edge coverage in order to generate test sequences non-vacuously.

Complete Path Coverage (CPC) involves covering all possible independent paths that exist in an automaton. A path starts from an initial node, then traverses through edges of the graph until it reaches a final node. Obtaining complete path coverage is infeasible if the graph contains any cycles since this can lead to infinite path lengths.

m-Path Coverage helps to remedy the infinite length problem in CPC. Usually it is desirable for generating test sequences (or paths) of a finite length while including all edges and cycles of the automaton. Both *Complete Path* and *Fixed Length* path coverages fall under the Path Coverage category in assertion coverages.

Edge Completion involves expanding an edge that contains a Binary expression of more than one variable into a set of edges that uniquely activates each variable. Figure 3.6 shows an automaton for three different transitions of Boolean operators. The first row shows a set of automata each containing a different primary Boolean operator between signals `a` and `b`. The second row depicts the *expanded* but equivalent versions of the automata, wherein the edge coverage covers each variable being fired.

The automaton in Figure 3.6(a) shows the condition `a&&b` to be at logic-1 for making the transition from state 0 to 1. Since the *and*-operation requires all Boolean variables be at logic-1, the expanded version of the automaton remains the same. For an *or*-operator in Figure 3.6(b), there are three possibilities for the original edge to become activated. In order to achieve complete *assertion variable coverage* by

Table 3.2: Relating Assertion and Automata Coverage

Assertion Coverage	**Automata Coverage**	**Vacuity Issue**
Covering all states of a property	Node Coverage	Yes
Covering all state transitions	Edge Coverage	No
Covering repeated Sequences of a property	Complete Round Trip Coverage	No
Combination of all of transitions	Complete Path Coverage	No
Assertion Step Coverage	Edge Coverage	No
Assertion Variable Coverage	Edge Completion + Edge Coverage	No

covering the edges, all the possible conditions must be exercised such that all variables were assigned to logic-1 or logic-0 at least once. Finally, in Figure 3.6(c) the edge completion for the *xor*-operator produces two outgoing edges. The defined automata coverages and their correlated assertion coverages are summarized in Table 3.2.

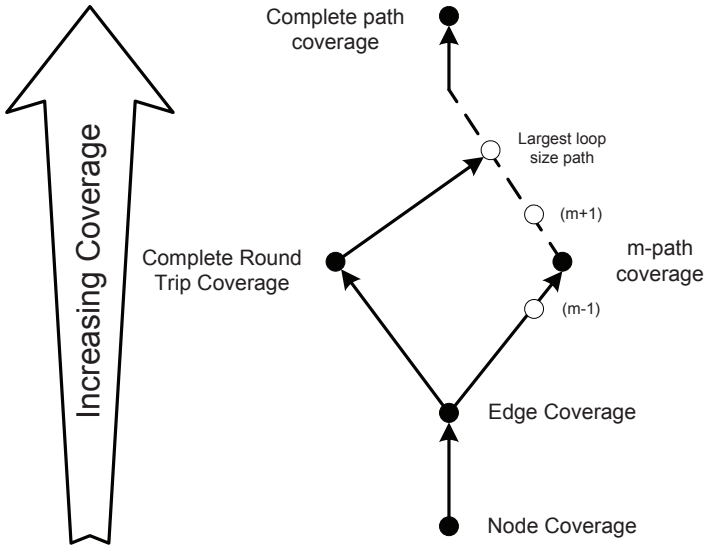
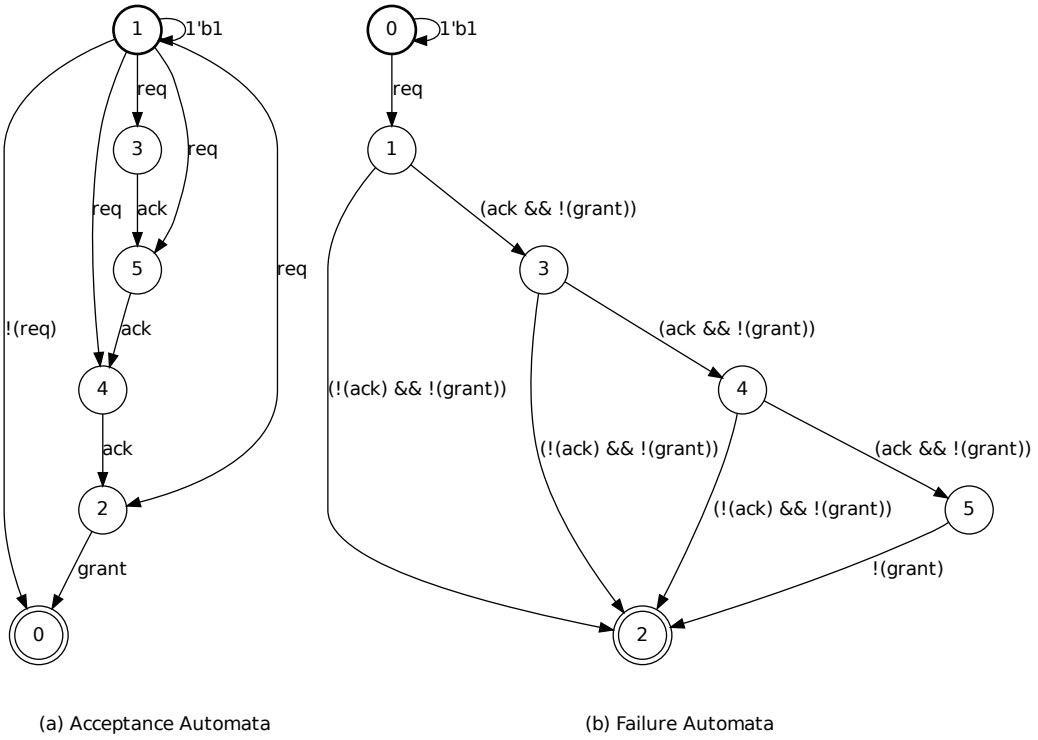


Figure 3.7: Partial ordering between different automata coverages

Partial order between the automata coverages is shown in Figure 3.7. Node coverage is completely subsumed in edge coverage since visiting all edges implies visiting all states (the reverse does not hold). The path coverage for the length of the largest loop



(a) Acceptance Automata (b) Failure Automata

Figure 3.8: Acceptance and failure automata for the example assertion : "assert property (@(posedge clk) req |→ ##1 (ack[*0:3]) ##1 grant);"

achieves CRTC. Below that length, the m-path coverage cannot be brought into the ordering relation with CRTC. Assertion variable coverage subsumes edge coverage, but is not comparable to other coverages.

3.3.4 Acceptance and Failure Automata Test Coverages

We mentioned that for test generation in ABV, the goal is to generate test sequences that can satisfy the expected behaviour of the design, as well as to exercise the scenarios that can potentially cause a fault. Shown in Figure 3.8 are the two types of automata based on the SVA assertion shown in the caption. For instance, the first automaton in Figure 3.8(a) shows the paths (or sequences) in order to exert the proper behaviour of the circuit. By efficiently traversing through all of the available paths in this automaton and covering all of the edges in order to generate non-vacuous test patterns, a test set of five vectors was derived.

However, for the failure automaton in Figure 3.8(b), an additional test set of four vectors was obtained. This implies that when solely using acceptance automata for generating test sequences, there is a possibility of achieving low test coverage. With

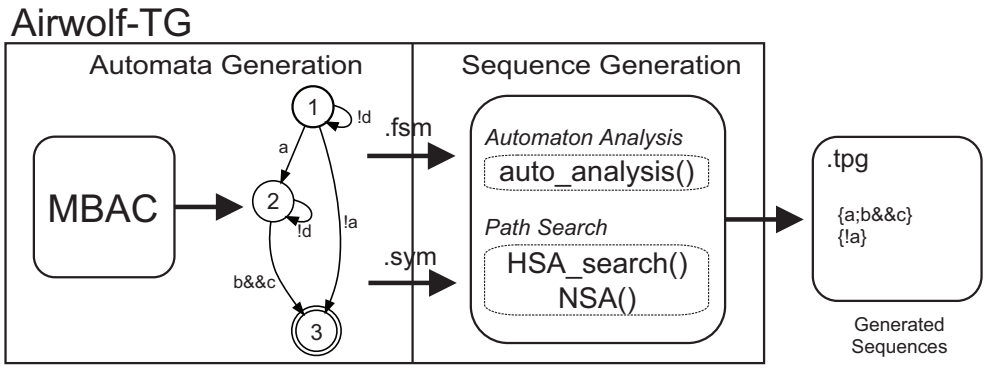


Figure 3.9: Test generation overview with MBAC and Airwolf-TG

additional test vectors generated using the failure automata, we attempt to exercise all of the correct and incorrect behaviour, and potentially increase the coverage of the entire test suite.

3.4 The Airwolf Test Generator

The objective of *Airwolf-TG* is to generate efficient test sequences that can either exercise the expected behaviour or a failure. The goal is to attain 100% coverage by applying the automata coverage metrics as shown in Table 3.2. Here are some of the constraints imposed on *Airwolf-TG*:

- Generating test sequences non-vacuously: As shown, this requires 100% *Edge Coverage* of an automaton where all edges are used at least once.
- Minimizing the reuse of edges that were traversed previously: This helps to reduce the amount of redundant edges that were already covered from a previous recursive search.

3.4.1 Test Generation Overview

Figure 3.9 shows *MBAC* and *Airwolf-TG* used together for generating test sequences for either acceptance or failure automata. In the first phase, *MBAC* produces efficient automata representations of the given properties, which in turn are used to model hardware checkers. A description of each automaton along with a list of its symbols are also produced, both of which are used by *Airwolf-TG*.

The sequence generation phase is split into two parts. Initially, an *Automaton Analysis* is performed by the function `auto_analysis`. This involves labelling the

Algorithm 3.1 Hybrid Search Algorithm for Automata Search

```
FUNCTION: HSA(𝒜(φ), Q_a, Π_In, Goal)
back_track ← 0, Q_new ← ∅
while q_c ≠ q_0 & back_track ≠ 1 do
  for all q_i ∈ Q_a do
    if q_i ∈ F /* If current state is a finish node */ then
      ext_path(Π_In[q_i]) /* Extract Test Path at current state */
    else
      Q_new = NSA(𝒜(φ), q_i, Π_In, back_track, Goal);
  if Q_new = ∅ then
    break
  else
    HSA (𝒜(φ), Q_new, Π_In, Goal)
    //Backtracking when returning from previous call
    back_track ← 1
return Π_φ
```

edges that can cause either non-deterministic behaviour, creating a cycle, or an edge that leads directly to a final node. Following this, the *Path Search* is called by functions `HSA_search` and `NSA`. These functions strategically explore the automata state space. The results from this phase are then stored into a separate file and ultimately form the sequences that cause a property to either pass or fail.

3.4.2 Airwolf-TG Algorithms

Airwolf-TG uses a combination of a modified Depth-First (DFS) and Breadth-First (BFS) searches to generate test sequences for any given finite automaton. These modifications were necessary for the following reasons: First, due the non-determinism, the tool should analyze the possibility of activating more than one successive node during the path search process. Second, since assertion automata can contain cycles, our algorithms check for them explicitly when minimizing the length of the test sequences of a property. Finally, our approach strategically chooses the subsequent set of edges to include as part of the test sequence based on the directional properties and *edge weight* calculations of each edge.

3.4.2.1 A Hybrid DFS/BFS Automata Search Algorithm

Algorithm 3.1 shows the pseudocode of the proposed *Hybrid DFS/BFS Automata Search Algorithm* (HSA). The `HSA` algorithm operates over a directed graph depicting the NFA representation of the assertion $\mathscr{A}(\phi)$, for which it receives a list of currently

Algorithm 3.2 Node Selection Algorithm (Node and Edge+CRTC Coverages)

```
FUNCTION: NSA (𝒜(φ), q_c, Π_In, back_track, Goal)
for all π_i ∈ Π_in do
  if q_c was not visited then
    if Goal = Node then
      Select edge e_j that was not previously traversed
    else if Goal = Edge & CRTC then
      Select edge e_j that is F/P/N NFA edge type
    E_sel ← E_sel ∪ e_j /* store selected edge */
    q_c ← as visited
  else if q_c was previously visited & back_track = 0 then
    if Goal = Node then
      Select edge e_j that was not travsered
    else if Goal = Edge & CRTC then
      if q_c has unused edges then
        E_sel ← E_sel ∪ E_qc /* Insert outgoing set of edges at q_c */
      else if q_c has an edge e_j that leads to q_d ∈ F /* Final State */ then
        E_sel ← E_sel ∪ e_j
Π_In ← Π_In∪E_sel
Q_new ← Subsequent nodes from outgoing edges stored in E_sel
return (Q_new)
```

activated states stored in the variable Q_a such that each state $q_n \in Q_a$ belongs to $\mathscr{A}(\phi)$, a coverage metric to apply for the test generation process (`Goal`), and an array of active incoming edges at each node Π_{In}.

The algorithm keeps track of the current active nodes at each recursive call of the `HSA` function. For determining the subsequent nodes to explore, each presently active node $q_n \in Q_a$ is sent to the *Node Selection Algorithm* for analysis and the decision to continue the forward traversal or not. Backtracking occurs when there are no subsequent nodes to explore. Finally, the `HSA` algorithm returns a set of paths, Π_ϕ describing the sequence of events for either passing or failing of assertion ϕ.

3.4.2.2 Node Selection Algorithm

Algorithm 3.2 shows the pseudocode of the *Node Selection Algorithm* (`NSA`). It receives from (`HSA`) a directed graph representing the NFA representation of the assertion ($\mathscr{A}(\phi)$), the current node (q_c), an array of active incoming paths (Π_{In}) of each node, a backtracking (`back_track`) flag, and finally *Goal* which defines the coverage metric to apply (Node/Edge+CRTC).

The `NSA` function is designed to generate test sequences for either *Node* or *Edge/CRTC*

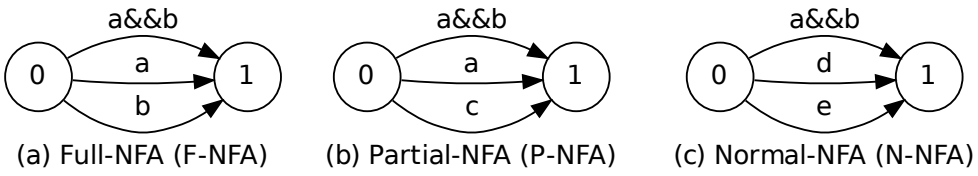


Figure 3.10: NFA node types

coverages. When choosing a specific *coverage goal*, the algorithm will try to explore all the possible paths by analyzing the edges' directional properties and Boolean expressions. Since the assertion automaton is non-deterministic, the `NSA` monitors which edge (or set of edges) can cause the automaton to enter into more than one successive state. Hence, during the automaton analysis, each node was labelled with a node type (*F-NFA, P-NFA, N-NFA*).

Shown in Figure 3.10 are the NFA node types. For a *F-NFA node* (Figure 3.10(a)), the `NSA` will select the edge that causes an activation of other edges of that node. In this case, all the edges are included. A *P-NFA node* (Figure 3.10(b)) occurs when one edge can activate a subset of edges coming from that same node while the dormant edges will be incorporated in later recursive calls. In the *N-NFA node* (Figure 3.10(c)), all its edges are uniquely activated and incorporated one at time.

During the backtracking phase, the `NSA` function tries to find any remaining unused nodes or edges to explore and traverse. When an unused node or edge is found, the forward traversal resumes. In this scenario, the search algorithm may encounter a set of nodes and edges that were previously explored. `NSA` tries to minimize the inclusion of redundant edges by finding an outgoing edge that leads directly to a final node ($q_d \in F$ – ending the test path), or selecting the edge of the least weight value (balancing the use of edges). At the end of the `NSA` call, the selected edges will then be assigned as the active incoming paths of each subsequent node stored into Π_{In}. Furthermore, those subsequent nodes will then be added to the new node list for the next recursive call.

3.4.2.3 Coverage Analyzer

A Coverage Analyzer (CA) is included to determine if a set of test vectors achieves the coverage goals. The algorithm plays a key role not only in determining the coverage, but also in combining the dissimilar methods, such as pseudo-random generation, with our scheme. Hence, it is of independent interest in the overall verification process, as we will detail when explaining the experimental results. Shown in Figure 3.11 is the

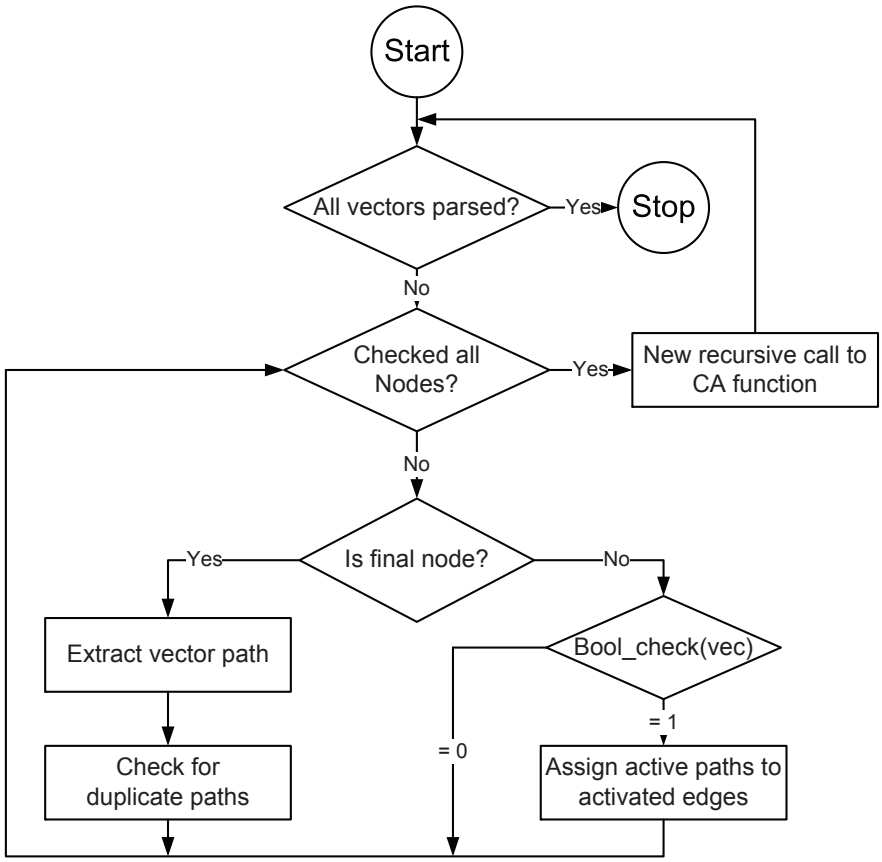


Figure 3.11: Flow chart of coverage analyzer function

flow chart of the CA analyzer.

The recursive function begins by checking all the current nodes in the current call. Each node will have its outgoing edges analyzed by the `Bool_check` function that determines a set of activated edges by the current vector `vec`. If `Bool_check` is true, the activated edges will then be assigned to the destination node as an active incoming path. Otherwise, CA continues to the next active node until all the nodes have been analyzed. If the CA encounters a final state, it will extract the path based on the current incoming active path assignments of each node. `Dupe_check` will determine if the path is already included in the vector set. Only distinct paths are kept which represent the unique path traversals that occurred in the automaton.

3.4.3 Run time and Correctness

Airwolf-TG's test generation process is a BFS/DFS combination search. When the nodes of the automaton are entirely of F-NFA type, the *Hybrid Search Algorithm* will behave in a Breadth-First manner as it searches all of the outgoing edges. On

the other hand, when all the nodes in the automaton have edges containing distinct Boolean expressions, and are strongly deterministic [103], the hybrid algorithm will then behave in a Depth-First manner as it searches a node at a time.

In the `HSA`, the computational portion of the function is cycling all the active nodes. At most, all the nodes of the automaton can be active concurrently which places an upper bound of N. When calling the `NSA`, the majority of the computation time is spent on assigning the active incoming path(s) to each node. If the edges of the automaton converge to a single non-final node, the largest number of active paths (including cycles) can potentially be all the edges of the automaton, denoted as E. Hence, the overall worst-case-scenario run time of our algorithm is $O(N \cdot E)$.

Regarding to the correctness, for every goal defined (Node or Edge+CRTC), the `NSA` algorithm finds all objects of the automaton and leaves nothing unused. Each traversed node/edge object becomes "marked" as visited upon a first encounter. The algorithm terminates when there are no unmarked objects remaining to visit. Additionally, for each traversal, a Boolean sequence is created by concatenating symbols for the transitions leading to all the objects. The sequence generated from an assertion automaton thus causes the traversal of the entire set of automaton objects of interest (edges, nodes), which satisfies the given coverage goal.

3.4.4 Test Sequence Generation Example

To illustrate our *Hybrid Search Algorithm*, we present an example based on the *MBAC* assertion automaton shown in Figure 3.12. The starting position of the hybrid-search begins at node 1. Two edges have the same Boolean expression (`sig1 && sig3`) which leads to subsequent nodes 0 and 2. Those nodes are added to the Q_{new} list which will be the next set of active nodes. Additionally, the edges that lead up to them are added into variable Π_{In} as the active paths, π_0 and π_2 respectively.

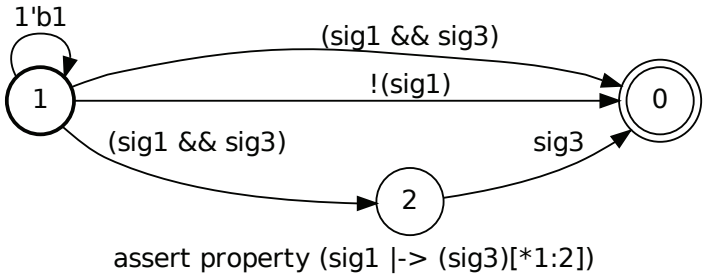


Figure 3.12: Acceptance automaton example

When the algorithm approaches the final node 0, it starts to output the test se-

quences. In this case, the edge between nodes 1 and 0 with the non-deterministic expression becomes extracted. The search resumes with node 2 by finding an outgoing edge to node 0 (`sig3`) that will be included into Π_{In}. The final node is once again added to the subsequent node list. Finally, the function extracts the Boolean sequences through the node path 1, 2 and 0.

Since there are no further nodes to traverse from node 0, the algorithm backtracks from node 2, and then to 1. It finds an unused deterministic edge between nodes 1 and 0 (`!sig1`). Eventually the algorithm will include that edge as part of the test sequence. In total, this example has generated 3 test sequences that cause the assertion to pass.

3.4.5 Coverage Analysis Example

The coverage analyzer in *Airwolf-TG* uses an approach similar to the *Hybrid Search Algorithm* for monitoring active nodes and edges. Instead of autonomously exploring the assertion space, a set of test vectors that accompanies the assertion can uniquely or non-determinstically activate certain edges of the automaton.

For the assertion automaton in Figure 3.12, the self terminating edge at node 1 is continuously active. Assuming the most and least significant bit of the binary test vector represent signals `sig1` and `sig3` respectively, as each test vector is read, the bits are analyzed to determine which variable is "high" or "low".

Assuming the first test vector {1,1} is read, this activates the edges between nodes 1 and 2 and nodes 1 and 0 both containing the Boolean expression: `sig1 && sig3`. A final node has thus been reached and the Boolean expression is extracted. CA will then determine if the sequence is a duplicate that was previously included, which in this case is false.

Currently, nodes 2 and 1 are active. A second vector {1,1} is sent to the CA, which detects that the edges between nodes 2 and 0 (`sig3`) and nodes 1 and 0 are activated. The extracted path between nodes 1,2 and 0 ({`sig1 && sig3; sig3`}) is retrieved and is then kept since it is not a duplicate. The path sequence between nodes 1 and 0 however, is rejected since it was already included.

A third vector {0,0} is then sent to the CA. The second path between nodes 1 and 0 (`!sig1`) is activated, whereas other edges remain dormant. Once again, the Boolean expression is extracted and compared to previously included test sequences. This entire process continues until all the test vectors are analyzed.

Table 3.3: Additional Coverage Relative to *MyGen*[2]

CPX ID	Total Vectors	Acceptance Vectors	Failure Vectors	Additional Coverage
0	5	60%	40%	66.7%
1	6	66.6%	33.3%	50%
3	5	60%	40%	66.7%
4	8	25%	75%	300%
6	4	25%	75%	300%
7	6	16.6%	83.3%	500%
9	11	91.6%	8.3%	9.1%
10	8	37.5%	62.5%	166.7%
12	19	52.6%	47.3%	90%
13	12	83.3%	16.6%	20%
15	2	50%	50%	100%
17	33	36.3%	63.6%	175%
19	22	27.2%	72.7%	267%

3.5 Experimental Results and Analysis

Three sets of experiments were carried out with *Airwolf-TG* from a set of *Complex* (CPX) properties that represent assertions used in industry [2]. Those properties are suitable for testing our tool since the automata produced by *MBAC* contain more paths and loops than the *Primitive* (PRIM) and *Limits* properties suite, also available at the URL link in [2]. As the CPX properties were written in PSL, we translate them into SVA and list them in Table 3.4. Some PSL assertions, however, do not have an SVA equivalent.

For the first experiment, we compared the amount of coverage attained using the Node coverage with the combined Edge/CRTC coverages on acceptance automata alone. The aim of the second experiment was an analysis of the test sequences generated with acceptance and failure automata. For the third experiment, we investigated if there were any differences in the number of unique test sequences generated between *MyGen* and *Airwolf-TG*. Generation of the two types of automata was performed by the *MBAC* tool, via the *failure* and *acceptance* modes.

In the first experiment, we generated test sequences with *Airwolf-TG* with emphasis on Node and Edge/CRTC coverages separately. Results are shown in Table 3.5. The automata coverage for each property was computed as a ratio between the covered (or traversed) edges to the total number of outgoing edges in the automaton.

Table 3.4: SVA Property Benchmarks (from *MyGen*[2], converted from PSL to SVA)

Prop ID	SV Assertion
CPX_SVA_0	property (@(posedge clk) (en_load && en_ud) |→ (sig1[→4] ##0 !sig2));
CPX_SVA_1	property (@(posedge clk) !resetg |→ (((!(sig1 & sig2))[*1:$]) |→ (en_load & en_ud)));
CPX_SVA_3	property (@(posedge clk) (sig1 → sig2) |→ ((!sig3 & !sig4)[*0:$] ##1 (sig3 & !sig4)));
CPX_SVA_4	property (@(posedge clk) ((sig1) or (sig2 ##1 sig3)) ##1 sig4);
CPX_SVA_6	property (@(posedge clk) ((sig2 ##1 sig3)[*4]) ##1 sig4);
CPX_SVA_7	property (@(posedge clk) (((sig1[*4]) ##1 (sig2))[*8]) ##1 (((sig3)[*0:$]) ##1 ((sig4 && sig5)[*2])));
CPX_SVA_9	property (@(posedge clk) !resetg |→ (##[1:10](en_load | en_ud)));
CPX_SVA_10	property (@(posedge clk) (sig1 ##1 sig2[*0:$]) |→ (sig3[*2] ##1 sig4));
CPX_SVA_12	property (@(posedge clk) !resetg |→ (sig1[→3:10] |→ (en_load | en_ud)));
CPX_SVA_13	property (@(posedge clk) !resetg |→ (sig1[→3:10] ##0 (en_load | en_ud)));
CPX_SVA_15	property (@(posedge clk) sig1 | sig2 | sig3 | sig4 | sig5 | sig6 | sig7 | sig8);
CPX_SVA_17	property (@(posedge clk) (((sig1[*0:$] ##1 sig2[*0:$] ##1 sig3[*0:$])) intersect (sig4[*5:7])) ##0 sig3[*0:$]));
CPX_SVA_19	property (@(posedge clk) sig1 |→ ((!sig5[*1:$]) \|→ (((sig2 ##1 sig3)[*0:$]) ##1 sig4[*3])));

Table 3.5: Node vs. Edge Coverage of Assertions using Acceptance Automata

CPX_ID	**Node Coverage Vectors**	% **Edge Coverage**	**Edge/CRTC Coverage Vectors**	% **Edge Coverage**
0	1	71.4%	3	100%
1	1	40%	4	100%
3	1	50%	3	100%
4	1	75%	2	100%
6	1	100%	1	100%
7	1	100%	1	100%
9	1	52%	11	100%
10	1	66.6%	3	100%
12	1	55%	10	100%
13	1	55%	10	100%
15	1	100%	1	100%
17	6	78%	12	100%
19	1	50%	6	100%

The majority of properties generated a single test sequence that visits all the nodes when using Node coverage; however, for some properties, the automata coverage is significantly lower compared to using Edge/CRTC coverage, where 100% automata coverage was achieved. These results show that Node coverage alone has the potential of leaving out sequences that can also pass or fail a property.

In the second experiment, we generated test sequences using acceptance and failure automata with *Airwolf-TG*. The set goal was to have 100% Edge/CRTC coverage, which avoids test sequences generated vacuously. Table 3.6 shows the number of vectors generated for each property when using acceptance and failure automata. The column "Total Vectors" gives the number of vectors produced by our tool for acceptance and failure automata combined. The "Min. Length" and "Max. Length" columns gives the length of the smallest and largest vector in the test set respectively. These numbers can be viewed as the number of clock cycles required to either pass or fail the property.

By observing the number of sequences generated using failure automata, we see an additional 3 to 6 vectors required to cause a property to fail. Additionally, properties like `CPX_SVA_`12, 17, and 19 present a significant difference. This implies that when considering only test patterns for verifying the proper behaviour of a design, there is a possibility that an unexpected behaviour during its execution or simulation may be

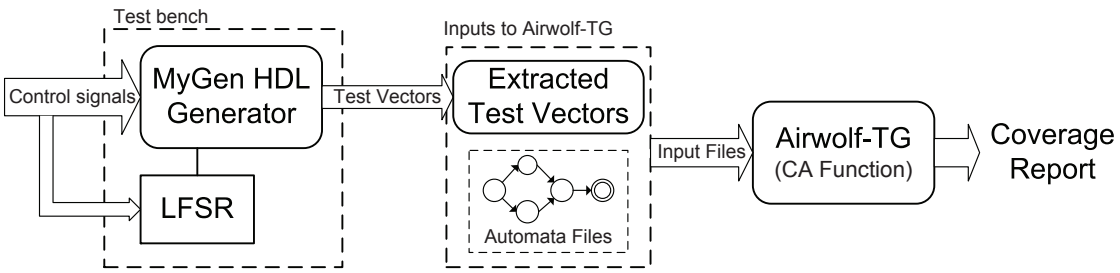


Figure 3.13: Coverage analysis process

perceived as being correct, when in fact it may be an incorrect response.

Since the *MyGen* tool [2] produces vectors only for the acceptance automata, in Table 3.3 we quantify the extra coverage obtained by our tool. The data is obtained by taking the acceptance and failure vector sets and is normalized such that 100% denotes both cases being taken into account. As shown in the table, a significant increase is observed over using only acceptance automata for generating test sequences, which is the case with the *MyGen* tool. For instance, the inclusion of failing states in examples such as `CPX_SVA_`12, 17 and 19, can contribute 47.3% to 72.7% of the test suite. Two properties such as `CPX_SVA_`9 and 13 are the only properties whose acceptance vectors contribute 80% or more of the test suite. These results indicate that generating acceptance test sequences alone may not contribute to a large portion of the coverage. When adding the failure test sequences, as we do in *Airwolf-TG*, additional vectors are available to try exploring the improper behaviour. In the case of design errors violating the properties, the incorrect response would manifest itself at the outputs upon applying these sequences.

In the third experiment, we used the Coverage Analyzer (CA) of *Airwolf-TG* for comparing the unique test sequences generated with *MyGen*'s hardware generators. Figure 3.13 depicts the entire process for this experiment. In the block diagram, the *MyGen* HDL generator is assisted by a Linear Feedback Shift Register (LFSR) that is used to generate pseudo-random test sequences. The entire entity is enclosed in a testbench file that provides stimulus to the generator and LFSR in order to capture the test vectors generated, which are then logged into a text file. The test bench extracted 1000 test vectors for each property.

The results retrieved by the `CA` function has shown little difference in the number of uniquely generated vectors between *MyGen* and *Airwolf-TG*. Both approaches have attained full edge coverage. There was a difference in the number of unique vectors for property `CPX_17`, in which *MyGen* generated 17 unique vectors whereas *Airwolf-TG* produced 12 vectors. When examining the path sequences that *MyGen* created, some

of the paths included previously traversed edges that were redundant compared to *Airwolf-TG*'s generated paths. The results presented show *Airwolf-TG*'s capabilities in generating test sequences compactly with respectable test coverage.

3.6 Summary

Assertion-based test generation is a much needed endeavour given the complex verification tasks, both present and future. In this chapter, a set of coverage coals were defined. These coverage goals were integrated into *Airwolf-TG*, a tool that generates test sequences that explore the specifications given by assertions. We derived a mapping from the assertion coverage goals to the coverage of the NFAs created by *MBAC* that represent the properties. We have presented methods that traverse the automata in order to generate efficient test sequences (based on a specified coverage metric), either from failure or acceptance automata. Our proposal is based on a hybrid search algorithm that is catered towards non-deterministic finite automata for run-time assertion checkers. Additionally, as automata may contain cycles that can create longer test sequences or activate more paths, the proposed algorithm minimizes the amount of loop traversals during the state space search.

With *Airwolf-TG* being able to efficiently traverse the automata, we compared the test vectors generated when using acceptance or failure automata on the same sets of properties. It was shown that some properties have generated additional test vectors that can potentially increase the overall test coverage, while having a modest increase in the test vector length. As a result, some of the benchmark assertions gave an increase of up to 70% of additional coverage, compared to when using acceptance automata alone. The additional tests sequences can then be used to effectively perform dynamic verification of digital designs.

Table 3.6: Acceptance versus Failing Sequences

		Acceptance Automata					Failure Automata				
ID	**Total Vectors**	**TG Generated**	**# of States**	**# of Edges**	**Min. Length**	**Max. Length**	**TG Generated**	**# of States**	**# of Edges**	**Min. Length**	**Max. Length**
CPX_0	5	3	6	12	1	5	2	6	11	4	5
CPX_1	6	4	3	7	1	2	2	3	5	1	2
CPX_3	5	3	3	6	1	2	2	3	5	1	2
CPX_4	8	2	4	5	2	3	**6**	5	10	1	3
CPX_6	4	1	10	10	9	9	**3**	10	12	1	9
CPX_7	6	1	43	44	42	42	**5**	44	50	1	44
CPX_9	12	11	12	22	1	11	1	12	12	11	11
CPX_10	8	3	5	8	1	4	**5**	5	10	1	4
CPX_12	19	10	12	31	1	11	**9**	12	30	3	11
CPX_13	12	10	12	31	1	11	2	12	23	10	11
CPX_15	2	1	2	2	1	1	1	2	2	1	1
CPX_17	33	12	14	26	5	5	**21**	19	39	1	7
CPX_19	22	6	7	14	1	7	**16**	9	27	1	8

Chapter 4

Test Compaction Techniques for Assertion-based Test Generation

Generating tests from NFA representations of assertions has the potential to create a large amount of tests. As discussed in Section 2.1.2, assertions are capable of defining complex behaviour that can be specified using temporal operators with large repetitions. This can increase the size of the NFA, which can lead to a large amount of tests for passing (or failing) an assertion.

This chapter presents *Airwolf-CTG*, a tool for generating compacted test sequences from assertions. The developed test compactor also relies on the generated tests from using NFA representations of assertions, which was presented with *Airwolf-TG* from Chapter 3. The proposed test compaction approach relies on grouping and exploiting similarities of multiple assertions. The similarities are used for finding redundant sequences of events that can satisfy (or falsify) more than one assertion. Additionally, the proposed test compactor can be integrated with another assertion-based test generator that was developed by another research team. The goal is to obtain a reduction in the size of the test set from assertions, thereby reducing the overall verification time.

4.1 Motivation

Assertion-based verification is gaining widespread usage among pre-silicon verification techniques as a powerful methodology for design verification [4]. Assertions are statements that explicitly define the intended behaviour of the Design Under Test (DUT). We use the fact that they can also serve as a *blueprint* for exploring the defined events of signals for satisfying the property. This behaviour exploration in-

cludes both testing in pre-fabrication verification and the monitoring of signals in the post-silicon phase [104]. Assertions, hence, provide the means of undertaking model-based directed test generation, wherein the temporal behaviour of the assertions can be used to drive the test generation process.

Using assertion-based directed test generation can potentially lead to large test sets. An assertion can model complex sequences of events that the DUT must produce, in turn generating many test sequences that can satisfy that same property. From our observations, combining multiple assertions with a certain level of similarity could help to produce overlapping sequences of events, which can then be shared. Exploiting these similar overlapping sequences can thus potentially reduce the test set size, which leads to a reduction in the overall verification time.

In this chapter, we present a novel method for generating a compacted set of test sequences by grouping similar assertions. Our approach assumes that all the Boolean signal conditions of every assertion references the primary inputs to the DUT. Each test vector applied to the DUT can thus be used to explore the events in multiple assertions concurrently. To assess our test compaction approach, we introduce the *Airwolf-CTG* tool (referred as *CTG* hereafter), which utilizes the standard non-deterministic finite automata approach [6] for generating test sequences. Our tool incorporates a set of proposed clustering methods that effectively group assertions with related sequences and signals. Then, our *CTG* tool employs two kinds of test compaction methods: *Test Path Overlapping* (*TPO*) and *Parallel-Path Removal* (*PPR*). We show that concurrently generating tests from multiple assertions can significantly reduce the test set size by as much as 98% when using *PPR* in some of the benchmarks. Additionally, our test compaction approach is also applicable for generating tests that cause an assertion failure, which further reduces the number of tests by as much as 86% when using *PPR*. The contributions that are made in this chapter are as follows:

- We introduce a set of clustering methods that groups assertions with common signals and sequences;
- We develop two test compaction algorithms for generating a reduced set of passing and failing test sequences for a set of assertions.

The contents of this chapter are organized as follows: [1] Section 4.2 presents our proposed test compaction methodology from assertions. From there, Section 4.3 shows

[1]The contents of this chapter is based on the article entitled *Test Compaction Techniques for Assertion-based Test Generation* [11] and the paper *Assertion Clustering for Compacted Test Sequence Generation* [12]

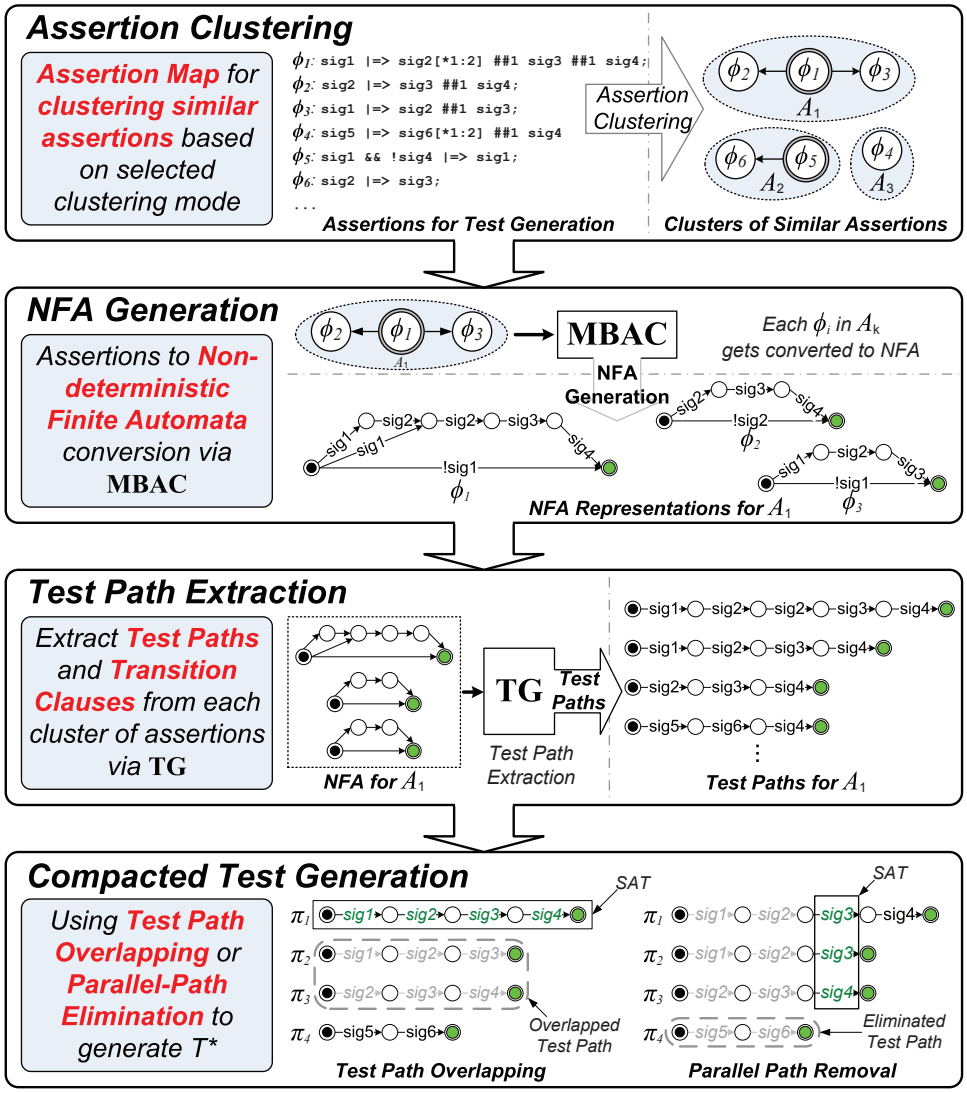


Figure 4.1: Proposed Test Compaction Methodology

our assertion clustering technique where we attempt to exploit as much similarities within different portions of the assertion itself. The test compaction techniques are presented in Section 4.4.

4.2 Proposed Compacted Test Generation Methodology

Figure 4.1 outlines the overview of our test compaction methodology. The first step is to divide the set of n assertions in A (the set of assertions) by using one of the four types of clustering methods, namely *antecedent*, *consequent*, *combined antecedent and*

consequent and *assertion signal clustering*. This generates a set of clusters C where each cluster $A_k \mid k = 0, 1, ...m$ holds a set of similar assertions.

Each cluster is sent to *MBAC* so that each assertion $\phi_i \in A_k$ can be transformed into its NFA representation, $\mathscr{A}$. As defined in Section 2.1.3, we use the *acceptance automata* representation of assertions for generating compacted test sequences that exercises the correct behaviour of the design. Our test compaction methodology is also applicable for *failure automata* when generating compacted tests that explores the incorrect behaviour of the design.

After NFA generation, the set of clusters C is then subjected to test path extraction using *TG*. The size of the NFA representation of assertions depends on temporal operators and sequences that were used in the assertion. This can imply that the generated set of test paths can be large. Each NFA representation of the assertion ($\mathscr{A}_{\phi_i}$) is sent to *TG*, which uses a heuristic that is based on a set of NFA coverage metrics [7] for generating and selecting test paths that are deemed significant. The coverage metrics guide *TG* in generating test paths that incorporates all transition clauses at least once while avoiding redundancies of previously included clauses, thereby creating a non-vacuous pass (or fail) of an assertion. *TG* stores the set of test paths into Π representing all the test paths for every assertion in cluster A_k.

Finally, each cluster of test paths undergoes compacted test generation which uses one of the proposed techniques, namely *Test Path Overlapping* (*TPO*) and *Parallel-Path Removal* (*PPR*), for generating a minimal number of tests for a similar set of assertions. The assertion clustering and compacted test generation are the main contributions in this chapter and will be thoroughly discussed in the subsequent sections.

4.3 Assertion Clustering

Assertion clustering is the process of dividing the set of assertions into disjoint groups that have a certain degree of similar signals and sequences. Clustering of similar assertions has been thoroughly studied with the intent of reducing the area overhead and power consumption when hardware assertions are grouped together. We incorporate assertion clustering into our framework so that we can generate a compacted set of tests for those grouped hardware assertion checkers, with the goal of reducing the overall verification time. Assertion clustering is performed on the assertions themselves where our algorithm analyzes the defined signals and sequences. On the other hand, performing clustering at the test path level would entail high costs

Algorithm 4.1 Assertion Clustering

FUNCTION: clustering
Input: A Set of Assertions (A)
Output: A Set of Clusters (C)

Step 1. Generate Assertion Map
for each $(\phi_i, \phi_j) \in A \mid i \neq j$ **do**
  $w_{i \to j} =$ **signal_sim**(ϕ_i, ϕ_j, *cluster_mode*)
  if $w_{i \to j} > \tau$ **then**
    Create edge $e_{i \to j}$ for (ϕ_i, ϕ_j) and include $w_{i \to j}$

Step 2. Compute σ for each ϕ_i
for each $\phi_i \in A$ **do**
  Compute σ_i of assertion ϕ_i
Sort set A based on descending σ_i value

Step 3. Cluster the similar assertions
k = 1 // Initialize Cluster ID
while $A \neq \emptyset$ **do**
  A_k = Group ϕ_i and its adjacent assertions
  Remove ϕ_i and its adjacent assertions from A
  $C \leftarrow C \cup A_k$, $k = k + 1$
return (C)

in computation time because every test path and their associated transition clauses must be analyzed when the size of the test paths and the number of referenced signals in each transition clause is large.

We assume that the assertions are in the form of `antecedent |=> consequent` which was defined from Section 2.1.2. This is a typical form that is usually seen in the vast majority of industry-based designs. Our goal is to generate a compacted set of test sequences that is tailored towards industry written assertions. We exploit as much similarity as possible within a cluster so that our compacted test generation algorithms are able to identify redundant test paths, thereby reducing the overall size of the test set.

4.3.1 Assertion Map and Similarity Weight

Assertion clustering is modelled by a weighted graph called an *Assertion Map*, that is generated during the first step of Algorithm 4.1. An Assertion Map $AM(A, E)$ is a directed graph containing a set of vertices A and a set of edges E. The set variable

A contains n assertions that are used for generating test sequences :

$$A = \{\phi_1, \phi_2, \phi_3, ..., \phi_n\}$$

where ϕ is a given assertion. This assertion set is the input to our clustering algorithm.

The set E of directed edges embodies the similarity between distinct pairs of assertions. Each directed edge $e_{i \to j}$ connects assertion ϕ_i to assertion ϕ_j in the AM provided they share a certain similarity. Each edge is thus labelled with a similarity weight $w_{i \to j}$. Any pair of nodes (ϕ_i, ϕ_j), where (i, j) $\leq n$ and $i \neq j$, has an edge $e_{i \to j}$ if and only if their computed similarity exceeds an overlap threshold value τ. The set of edges is therefore as follows :

$$E = \{e_{i \to j}\} \mid w_{i \to j} > \tau, i \neq j \tag{4.1}$$

The threshold value τ is used in the clustering algorithms to generate clusters with that minimum amount of similarity.

In computing the similarity weight for any pair of assertions, the clustering algorithm first determines the number of similar signals between assertions ϕ_i and ϕ_j which is computed by the `signal_sim` function. The set of similar signals is defined as $S_{i \leftrightarrow j}$, and its size denoted as, $|S_{i \leftrightarrow j}|$. The exact criteria for determining which signals are deemed to be similar depends on the clustering mode employed (`cluster_mode` in the algorithm). The next subsection presents multiple ways in which such criteria can be implemented. Once a clustering mode is selected, the similarity weights are computed as follows :

$$w_{i \to j} = \frac{|S_{i \leftrightarrow j}|}{|S_j|} \tag{4.2}$$

which is the ratio between the number of similar signals between a pair of assertions ϕ_i and ϕ_j to the total number of signals defined in ϕ_j. It is important to note that both $w_{i \to j}$ and τ have values between 0 and 1. For example, if $w_{i \to j} = 1$ for the *antecedent clustering* mode, this implies that the signals defined in assertion ϕ_i match all the signals defined in ϕ_j.

After computing the similarities of all the assertions, the clustering algorithm then determines the sum of all edge weights σ_i for each node i (i.e. assertion ϕ_i) as follows :

$$\sigma_i = \sum_{i \neq j} w_{i \to j} \mid w_{i \to j} > \tau \tag{4.3}$$

This is performed in Step 2 of the algorithm.

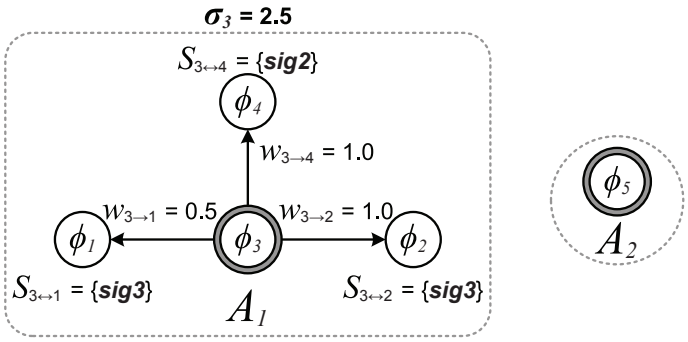


Figure 4.2: Antecedent Clustering. In the left cluster, the computed similarity weights are shown in the edges, and the sum of weights for ϕ_3 is 2.5 (σ_3)

Thereafter, the node with the largest weight summation will cluster its adjacent nodes (similar assertions) together, thereby removing those nodes from further clustering (shown in Step 3). The size of each cluster can vary between having as many individual clusters with a single assertion to a single cluster with the entire assertion set. These clusters of assertions are then used by our compacted test generation algorithm.

4.3.2 Clustering Modes

In this section, we describe our proposed clustering methods namely: *antecedent clustering*, *consequent clustering*, *combined antecedent and consequent clustering* and *assertion signal clustering*. The paragraphs that follow discuss how each clustering mode computes the similarity weights of the assertions and how they affect the clusters that are generated.

Antecedent Clustering involves searching for similar signals in the left part of the temporal implication operator `|=>`. For example, consider the set of assertions defined below :

ϕ_1: `sig1 ##1 sig3 |=> sig2 && sig4 ##1 sig5`
ϕ_2: `sig3 |=> sig2 ##1 sig5;`
ϕ_3: `sig2 && sig3 |=> sig6 && sig4 ##1 !sig7;`
ϕ_4: `sig2 |=> sig3 || sig6;`
ϕ_5: `sig5 && sig3 ##1 sig4 |=> sig1;`

The clustering algorithm analyzes each assertion and determines if the number of

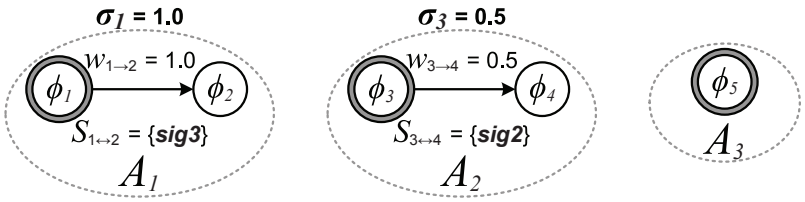


Figure 4.3: Consequent Clustering

signals defined in the antecedent portion is able to exceed the required overlapping threshold on signals defined in other assertions. If we define a threshold value of τ = 0.5 for antecedent clustering, we see that the assertion ϕ_3 is able to overlap all of the signals defined in assertions ϕ_2 and ϕ_4. This is due to the signals `sig3` and `sig2` both overlapping the signals in the antecedent in ϕ_2 and ϕ_4, respectively. Thus, the set of similar signals between these assertions with respect to assertion ϕ_3 are $S_{3\leftrightarrow2} = \{\texttt{sig3}\}$ and $S_{3\leftrightarrow4} = \{\texttt{sig2}\}$. Using equation (4.2) yields a similarity weight of 1.0 for $w_{3\rightarrow2}$ and $w_{3\rightarrow4}$. Similarly, the signals in the antecedent of assertion ϕ_3 also overlaps assertion ϕ_1, thereby having a similar signal set $S_{3\leftrightarrow1} = \{\texttt{sig3}\}$. Then, applying equation (4.2) yields a similarity weight of $w_{3\rightarrow1} = 0.5$. Assertion ϕ_5, has the least number of signals that are similar with other assertions due to the fact that signals `sig4` and `sig5` were not defined in other assertions, thus achieving a similarity weight of $w_{3\rightarrow5} = 0.33$ which is below the required threshold value. After completing the weight similarity computations, the clustering algorithm computes the summation of each node's surrounding similarity weights using equation (4.3). As depicted in the figure, assertion ϕ_3 has the largest weight total $\sigma_3 = 2.5$, while assertion ϕ_5 had the least similarity. Thus, the clustering algorithm generates two clusters which are $A_1 = \{\phi_1, \phi_2, \phi_3, \phi_4\}$ and $A_2 = \{\phi_5\}$.

Consequent Clustering attempts to group assertions that have similar signals defined in the right part of the implication operator. There are cases in which assertions can have overlapping signals and sequences in the consequent, and grouping them can potentially benefit other assertions.

The clustering algorithm begins by analyzing the signals in all of the assertions' consequent portions. Signals `sig2`, `sig4` and `sig5` from the consequent of assertion ϕ_1 overlap all the defined signals in the consequent of ϕ_2, thus creating a set of similar signals $S_{1\leftrightarrow2} = \{\texttt{sig2}, \texttt{sig5}\}$ and a similarity weight of $w_{1\rightarrow2} = 1.0$. The consequent portion of assertion ϕ_3 overlaps half of the signals defined in the consequent of assertion ϕ_4. Thereby having the set of similar signals and similarity weight $S_{3\leftrightarrow4}$ = $\{\texttt{sig3}\}$ and $w_{3\rightarrow4} = 0.5$, respectively. Assertion ϕ_5 did not have any similar signals

with other assertions, thus it is not connected to other nodes in the AM. When summing the similarity weights of all the nodes (assertions), the clustering algorithm generates three clusters which are $A_1 = \{\phi_1, \phi_2\}$, $A_2 = \{\phi_3,\phi_4\}$ and $A_3 = \{\phi_5\}$, as shown in Figure 4.3.

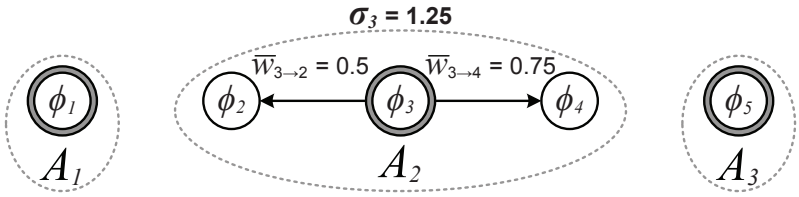


Figure 4.4: Antecedent and Consequent Clustering

Combined Antecedent and Consequent Clustering takes the weight values w that were computed using *antecedent* and *consequent* clustering modes in each assertion and computes the average, $\overline{w}$. This average value determines the overall similarity of other assertions in both the antecedent and the consequent portion. From the same assertion set with the unchanged threshold value of τ, the clustering algorithm computes the average similarity weights for all the assertions in the AM. Assertion ϕ_3 has an average similarity weight ranging between assertions ϕ_2 and ϕ_4, which are $\overline{w}_{3\to2} = 0.5$ and $\overline{w}_{3\to4} = 0.75$, respectively. Assertion ϕ_1 has overlapping signals defined in the antecedent and consequent of assertion ϕ_2 and the average similarity weight is $\overline{w}_{1\to2} = 1.0$. The same can also be said for assertion ϕ_5, which has a computed average similarity weight with ϕ_2 of $\overline{w}_{5\to2} = 0.5$.

After the average similarity weights are computed between all the assertions, assertion ϕ_3 has achieved the largest weight sum of $\sigma_3 = 1.25$ with assertions ϕ_2 and ϕ_4. Thus, three clusters are generated: $A_1 = \{\phi_1\}$, $A_2 = \{\phi_2, \phi_3,\phi_4\}$ and $A_3 = \{\phi_5\}$, as depicted in Figure 4.4.

Assertion Signal Clustering takes each assertion and computes the similarity of the uniquely defined signals as a whole. For example, if we attempt to compute the similarity of assertions ϕ_1 and ϕ_2, we determine the signal set of ϕ_1 as (`sig1`, `sig2`, `sig3`, `sig4`, `sig5`) and ϕ_2 as (`sig2`, `sig3`, `sig5`). We see that the signals defined in ϕ_1 overlap all of the uniquely defined signals in ϕ_2 and ϕ_5. Thus, the similar signal set becomes $S_{1\leftrightarrow2} = \{$`sig2`, `sig3`, `sig5`$\}$ and $S_{1\leftrightarrow5} = \{$`sig1`, `sig2`, `sig3`, `sig4`$\}$. The similarity weights in $w_{1\to2}$ and $w_{1\to5}$ are both equal to 1.0.

Similarities can also be found in assertions ϕ_3 and ϕ_4 with respect to assertion ϕ_1. The set of similar signals between these assertions then become $S_{1\leftrightarrow3} = \{$`sig2`, `sig3`, `sig4` $\}$ and $S_{1\leftrightarrow4} = \{$`sig2`, `sig3`$\}$, with their computed similarity weights being $w_{1\to3}$

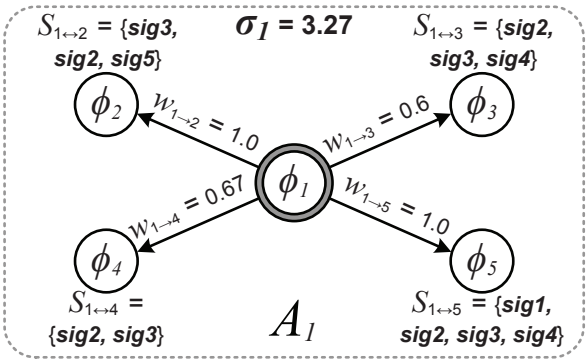


Figure 4.5: Assertion Signal Clustering

= 0.6 and $w_{1\to4} = 0.67$, respectively. Assertion ϕ_1 therefore has a weight similarity total of $\sigma_1 = 3.27$, which then produces the cluster $A_1 = \{\phi_1, \phi_2, \phi_3, \phi_4, \phi_5\}$.

The proposed clustering methods attempts to group assertions with various signals defined in either the antecedent or the consequent parts. Depending on the nature of the assertions, some of the clustering methods can produce optimal clusters with significant overlap. In the next section, we discuss our test compaction approaches.

4.4 Compacted Test Sequence Generation

The clusters of assertions generated from our clustering algorithm are used for generating a compacted set of test sequences. Each cluster is utilized by one of our two compacted test sequence generation algorithms, namely *Test Path Overlapping* (*TPO*) and *Parallel-Path Removal* (*PPR*). Both compaction algorithms maintain the order of events that are defined in every test path, π_i, which creates either an assertion pass or fail. Additionally, they ensure that the value of a Boolean signal does not conflict with others during test compaction [56]. Furthermore, the PPR algorithm takes advantage of the unspecified signal conditions that is similar to the approach employed in [55]. This allows the combining of multiple transition clauses to form a single test, thereby having multiple test paths reaching to their final states. These concepts were originally developed for compacting manufacturing tests; however, we apply them for compacting tests for functional verification. In this section, we describe our proposed compacted test sequence generation algorithms where each approach is discussed in detail followed by an illustrative example.

Algorithm 4.2 *Test Path Overlapping*

FUNCTION: TPO
Input: A Cluster of Similar Assertions (A_k)
Output: Test Set (T^*)

{Phase 1: Test Path Extraction}
for each $\phi_i \in A_k$ **do**
1. $\mathscr{A}_{\phi_i} \leftarrow$ **MBAC**(ϕ_i)
2. $\Pi \leftarrow \Pi \cup$ **TG**$_\Pi(\mathscr{A}_{\phi_i})$
3. $\Psi \leftarrow \Psi \cup$ **TG**$_\Psi(\mathscr{A}_{\phi_i})$

{Phase 2: Test Compaction using TPO}
4. **Sort**(Π) by descending lengths

while $\Pi \neq \emptyset$ **do**
$\pi_{base} \leftarrow$ largest $(|\pi| \in \Pi)$, *Remove* π_{base} from Π
5. $T_s \leftarrow$ **SAT**(π_{base})

{*Redundant Test Path Removal*}
for each $\pi_i \in \Pi$ **do**
if T_s evaluates all clauses in Ψ_{π_i} to true **then**
6. Remove π_i from Π

7. $T^* \leftarrow T^* \cup T_s$

return (T^*)

4.4.1 Test Path Overlapping

Algorithm 4.2 shows the proposed *Test Path Overlapping* approach which is based on the SAT-solution sharing concept from [64]. The novelty of our approach is to generate test sequences by running a satisfiability (SAT) solver on a set of transition clauses Ψ within a test path π_i (denoted as Ψ_{π_i}). We use SAT-solution sharing for identifying and removing redundant test paths within a cluster of similar assertions, thereby potentially reducing the overall size of the test set. The *TPO* algorithm takes in a cluster of similar assertions, A_k, and is subjected to two phases. The first phase is test path extraction and their collection of transition clauses from each assertion. The second phase performs the compacted test sequence generation where the Boolean sequences are generated by a SAT solver, followed by the removal of redundant test paths. Following these two phases, the algorithm returns a set of compacted tests, T^*, which represents the test sequences used for satisfying the similar set of assertions.

Phase 1 begins by transforming each assertion, $\phi_i \in A_k$, into its NFA representation $\mathscr{A}_{\phi_i}$ using **MBAC**. In the second step the test paths from $\mathscr{A}_{\phi_i}$ are extracted by the function $\mathbf{TG}_{\Pi}$ and stored into the set variable Π. Each test path π_i contains a sequence of transition clauses, Ψ_{π_i}, which represent the Boolean conditions required for reaching a final state. In step 3 each test path's transition clauses are extracted by the function $\mathbf{TG}_{\Psi}$ and stored into the set variable Ψ.

Phase 2 of the algorithm uses the extracted test paths and transition clauses from phase 1 for generating compacted test sequences for a cluster of similar assertions. The second phase begins by sorting all the test paths in descending order according to their lengths ($|\Psi_{\pi_i}|$). The intention is to use the longest test path as the base path, π_{base}, in order to perform SAT solving on a potentially large set of transition clauses, which can then be shared with other test paths within the cluster. After finding the longest path, it is then removed from the test path set, Π, so that it will not be considered for further analysis.

The base test path gets sent to the SAT solver which determines the appropriate assignments of Boolean logic values on the set of transition clauses, $\Psi_{\pi_{base}}$. Results obtained are then stored into the set variable, T_s. The sequence of Boolean logic values generated by SAT become the test sequences that are used for satisfying an assertion. It is important to note that the ordering of these test sequences must be respected since they were generated based on the order of appearance of the transition clauses of a test path. This sequence ordering causes the test path to enter its final state, thereby creating a successful pass through the assertion.

After generating test sequences from the base path, the algorithm attempts to find redundant test paths in Π. To perform this step, a search is performed by analyzing each of the test paths' set of transition clauses, Ψ_{π_i}. The algorithm then uses the current base path's test sequences stored in T_s and determines if the ordering of those tests can continuously evaluate the entire set of transition clauses in Ψ_{π_i} to true. This is performed by taking the assigned Boolean logic values of each sequence and determining if the clause evaluates to either true or false. If the entire set of transition clauses evaluates to true, then test path π_i is deemed redundant since it can enter its final state based on the tests generated from the base path, π_{base}. As a result, the algorithm removes the redundant test path which can potentially reduce the overall number of tests. Otherwise, the test paths remains in the set and will be analyzed in the next iteration. Finally, the current test sequences are stored into T^*. The algorithm repeats these steps until there are no test paths remaining for analysis and test generation. When this occurs, the algorithm proceeds to the next cluster of

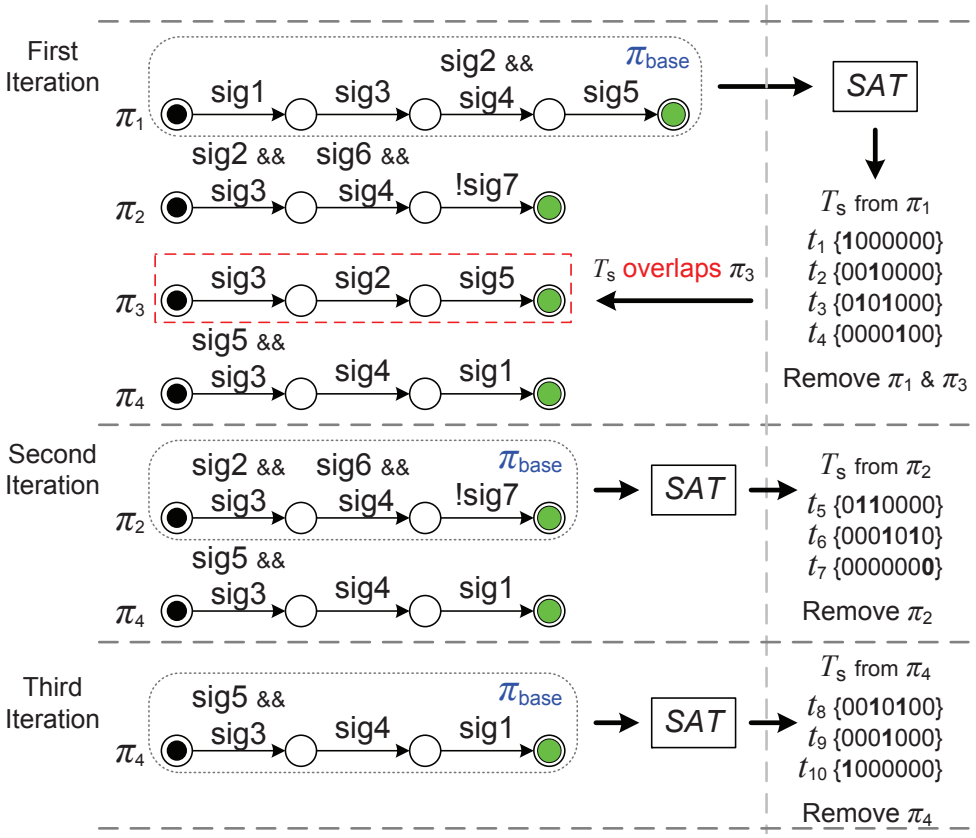


Figure 4.6: Test Path Overlapping Example

assertions.

4.4.2 *TPO* Example

Figure 4.6 shows a sample three iteration run of our *TPO* algorithm. Each bit in the test represents {sig1,sig2,sig3,sig4,sig5,sig6,sig7}. There are four sample test paths which were generated from a cluster of assertions in A_1 when using the antecedent signal clustering mode. The test paths were sorted in descending order of length.

The first iteration shows that test path π_1 is the current base path which is enclosed inside the dashed box as seen in the figure. The base path contains four transition clauses that describe the required conditions for reaching its final state. Each transition clause $\psi_i \in \Psi_{\pi_{base}}$ is converted into its DIMACS representation that can be used for SAT solving. Since there are four transition clauses inside the current base test path, then four test sequences will be generated and stored into T_s.

Each test sequence in T_s contains the appropriate Boolean logic values that are assigned to each signal. The signals defined for each test sequence within this cluster

are {`sig1`, `sig2`, `sig3`, `sig4`, `sig5`, `sig6` and `sig7`}. For instance, the third transition clause of the base path is `sig2 && sig4` which requires signals `sig2` and `sig4` to be at logic-1. This causes `sig2` and `sig4` to have a value of a logic-1 as depicted in t_3. Signals that were not included in the transition clause are treated as don't care values; however, we assign them to logic-0 since we only consider the Boolean values of the signals defined inside a transition clause.

The test sequences stored in T_s are then used for identifying redundant test paths in Π. This is done by determining if the ordering of current test sequences generated from π_{base} can cause other test paths in the set to enter into their final states. For example, in test path π_2, the first transition clause is ψ_1 =`sig2 && sig3`, which requires a value of logic-1 under the signals `sig2` and `sig3`. Since there was no test in T_s that satisfies this condition, then this first clause of π_2 is evaluates to false, causing the algorithm to analyze the next test path.

Test path π_3 was able to reach its final state. This is due to the fact that the test sequence from t_2 to t_4 causes the entire set of transition clauses in Ψ_{π_3} to evaluate to true continuously. This implies that the test sequences generated from the current base path (π_1) subsumes the test path of π_3. Thus, π_3 is deemed redundant and consequently removed from Π, thereby reducing the number of tests generated. This is the key principle behind the *TPO* test compaction approach.

Test path π_4 was not able to evaluate all of its transition clauses to true continuously. The test set from T_s did not have a test containing a logic-1 under the specified signals. Thus, this test path remains in Π. At the end of the first iteration, the algorithm will add T_s to T^*. Then, it removes test paths π_1 and π_3 from Π.

The second iteration begins by selecting test path π_2 as the base path. Performing SAT-solving on the set of transition clauses Ψ_{π_2} generates three test sequences that are listed as t_5, t_6, t_7 in T_s. Then, the algorithm will attempt to identify any redundant test paths based on the sequences generated from π_2; however, the tests in T_s were unable to cause the transition clauses in π_4 to continuously evaluate to true. Since there are no remaining test paths to analyze at this current iteration, the algorithm will add T_s to T^* and remove test path π_2.

At the third iteration, the remaining test path π_4 is selected as the base path for test generation. Finally, after π_4 is removed, there are no more test paths and the test compaction is finished. Thus, a total of 10 test sequences were generated and stored in T^*.

Algorithm 4.3 *Parallel-Path Removal*

FUNCTION: PPR
Input: A Cluster of Similar Assertions (A_k)
Output: Test Set (T^*)

{Phase 1: Test Path Extraction}

{Phase 2: Test Compaction using PPR}
Sort Π by descending lengths

while $\Pi \neq \emptyset$ **do**
  $n_t \leftarrow$ largest $|\pi_i| \in \Pi$
  $\Pi_r \leftarrow 0$
  for $j = 0 \rightarrow n_t$ **do**
    $\Psi_s \leftarrow \emptyset$
    for $\pi_i \in \Pi$ **do**
      if add_clause(Ψ_s, ψ_j) = false **then**
        $\Pi_r \leftarrow \pi_i$
      if $\psi_j \in \Psi_{\pi_i} \mid (s, \psi_j, d), d \in F$ **then**
        Remove π_i from Π
      $T^* \leftarrow T^* \cup$ **SAT**(Ψ_s)

  if $\Pi_r \neq \emptyset$ **then**
    $\Pi \leftarrow \Pi_r$

return (T^*)

4.4.3 Parallel-Path Removal

The second proposed algorithm, namely *Parallel-Path Removal* (*PPR*), generates compacted test sequences by analyzing transition clauses from every test path concurrently. This test compaction algorithm is based on the test set relaxation concept that assigns a Boolean value to the *don't care* bits of a test in order to increase the number of detectable faults. We use this concept with our compaction algorithm so that a generated test can cause multiple test paths to enter into either their next or final states, thereby potentially passing multiple assertions.

Algorithm 4.3 shows our proposed *PPR* compaction algorithm. The first phase of the algorithm begins by extracting a set of test paths and the associated transition clauses from a set of similar assertions, A_k. The tasks performed in this first phase are the same as those described in Algorithm 4.2.

The second phase of the *PPR* algorithm begins by finding the longest test path

length and stores that value into the variable, n_t. This compaction approach employs *a sweep* of transition clauses of all the test paths in the set Π. The purpose is to analyze each transition clause starting from the initial state ($j = 0$) to the final state of every test path ($\pi_i \mid i = 0, 1, ..., |\Pi|$). This ensures that every test path of length less than n_t gets analyzed and used for test generation.

Set variable Ψ_s is used to store the j^{th} transition clauses from every test path. We define the function `add_clause` which takes as input the set of transition clauses Ψ_s and a transition clause ψ_j from a test path π_i. It is important to note that the stored transition clauses must have a single logic value assignment for every signal defined. For example, if a transition clause j from test path π_a is $\psi_j = (b_1$ `&&` $b_2)$ and in test path π_b is $\psi_j = (!b_1)$, then these clauses are unable to have a single logic value assignment for b_1. If the `add_clause` function returns a false value, this implies that the transition clause ψ_j at test path π_i cannot be added into Ψ_s. The algorithm removes π_i from the current iteration and gets stored into set variable Π_r, which stores the test paths for analysis at the next test compaction iteration.

After completing the analysis of all test paths at the j^{th} transition clause, then the next step is to determine if $\psi_j \in \Psi_{\pi_i}$ causes a test path π_i to enter into its final state. If this condition is true, then this implies that all the clauses from Ψ_{π_i} were successfully used for test generation, thereby causing the algorithm to remove test path π_i from further analysis. This is the crux of the *PPR* algorithm.

The transition clauses in Ψ_s get sent to SAT for determining the appropriate Boolean value assignments of every defined signal. Results obtained from SAT(Ψ_s) is stored into T^*. The analysis continues to the next transition clause ψ_{j+1} and ends when j reaches to value n_t (longest test path).

After all the paths have been analyzed in Π, the algorithm determines if there are any remaining test paths in the set variable Π_r. *PPR* continues its test compaction approach if Π_r has remaining test paths, otherwise execution ends and then returns a set of compacted tests, T^*.

4.4.4 *PPR* Example

Figure 4.7 is an example run for using *PPR* compaction on the four test paths. The set of test paths Π are sorted in descending order according to the number of transition clauses. As depicted in the figure, test path π_1 is deemed to be the longest in this set. Thus, the value of the variable n_t is 4.

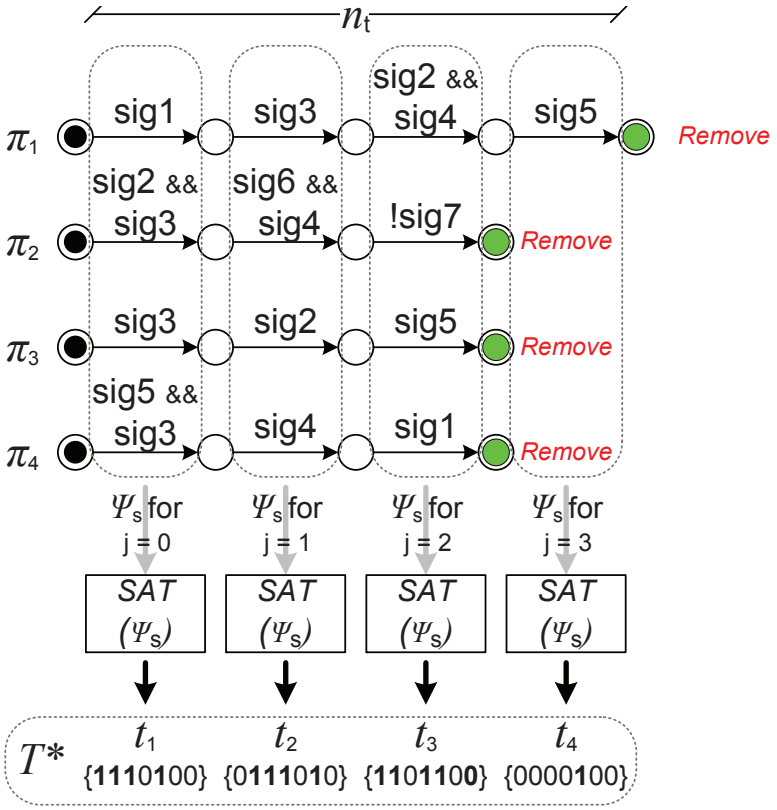


Figure 4.7: Parallel Path Removal Example

PPR begins by analyzing the first transition clause, $j = 0$, from all of the test paths in Π. The first transition clause in test path π_1 is $\psi_0 =$ `sig1`, and since the set variable Ψ_s is empty, ψ_0 from π_1 is added to Ψ_s. Next, the first transition clause in π_2 is $\psi_0 =$ `sig2 && sig3`, which gets sent along with the set variable Ψ_s to the function `add_clause`, which determines if the defined Boolean variables have only one logic value assignment. Since there are no shared variables between these clauses, then the transition clause is added to Ψ_s.

Next, the first transition clause from the third test path is passed to `add_clause` for analysis. The signal `sig3` is already included in Ψ_s, therefore requiring a Boolean value of logic-1. For this test path, the Boolean conditions defined in ψ_0 require `sig3` to be at logic-1 which satisfies the previous condition from the second test path. Thus, the transition clause `sig3` gets added to Ψ_s. This reasoning is also applied to ψ_0 from the fourth test path, where the Boolean condition requires signal `sig3` to be at logic-1. Since this satisfies the other previous Boolean conditions from other test paths, the transition clause `sig5 && sig3` gets added to Ψ_s.

At the end of every function call to `add_clause`, the *PPR* algorithm determines if

the current test path π_i has reached the final state. Since every test path has entered its next state, which does not belong to the set of final states, then those test paths remain for further analysis.

The algorithm now passes the set variable Ψ_s, which contains all the j^{th} transition clauses ($j = 0$ for this sample run) that gets sent to SAT for generating a single test t_i. Each test contains the appropriate logic values that causes *multiple test paths* within a cluster of assertions to enter into its next state concurrently. This is the crux behind the *PPR* algorithm for compacting test sequences from assertions. Results retrieved from SAT are stored into T^*. The algorithm continues to the next ψ_{j+1} transition clause and finishes until j reaches the value of largest test path length, n_t.

The test sequences generated with the *PPR* algorithm are different from those using the *TPO* scheme. In the *PPR* algorithm, a test is generated at the end of analyzing the j^{th} transition clause from every test path. Each test contains Boolean value assignments that causes multiple test paths to enter into their next or final states. The *TPO* algorithm generates tests from a single test path. Those same tests sequences are used to find redundant test paths within the same set of assertions, although not as effectively as analyzing all the test paths concurrently during the test generation phase. For instance, $t_1 = \{1110100\}$ has signals `sig1`, `sig2`, `sig3` and `sig5` assigned to logic-1. These logic value assignments cause the entire set of test paths to enter into the next state. Similarly, test $t_2 = \{0111010\}$ has signals `sig2`, `sig3`, `sig4` and `sig6` assigned to logic-1 concurrently. Thus, they also cause all the test paths to enter into the next state. Finally, in test $t_3 = \{1101100\}$ contains the appropriate Boolean values for causing test paths π_2 to π_4 to enter into their final state. Thus, three test sequences were required for creating a passing result of three assertions within the same cluster.

The above example shows that the *PPR* algorithm was able to satisfy a similar set of assertions with only four test sequences. In comparison, the *TPO* compaction scheme requires 10 test sequences in order to satisfy this same set of assertions. This shows a significant improvement in terms of the number of tests generated when using the *PPR* compaction method.

4.5 Experimental Results

To verify the effectiveness of our proposed compacted test generation methodology, we have considered two case studies, each comprised of five different sets of assertions. In the first case study, we evaluate our *CTG* tool for compacting good (passing)

and failing test sequences by using the four proposed clustering modes and the two compaction approaches. Our second case study is to use *MyGen* [2] for generating and compacting passing and failing test sequences. The work of *Oddos et al.* is closely related to ours as they also use NFA representations of assertions for producing tests. We use these tests for determining which test paths in Π are overlapped, thus contributing to a reduction in the test set size, $|T^*|$. The goal of both studies is to gauge the amount of compaction that was achieved compared to no test compaction applied with both tools.

Five sets of assertions were used in each case study. Each set contains assertions that were used to verify industry standard protocols and interfaces. The first set is ARM's AMBA 3 High Performance Bus (AXI) [9] which contains a suite of assertions for verifying the timing requirements of the design's control signals when it is using the AXI interface. The next two sets are synchronous (FF-SC) and asynchronous (FF-GC) FIFO assertions that are provided by PMC-Sierra. These assertions were written in-house for verifying the correct functionality of the two FIFO controllers used for interfacing with other circuits. Lastly, we used two sets of assertions that are available from [105], namely the Peripheral Component Interconnect (PCI) protocol and the SDRAM controller, which are both used to verify the correctness of the control signals' timing requirements. Since these assertions were written for industry standard protocols, it would be beneficial to attempt to compact the generated tests, thereby accelerating the conformance testing of the design.

4.5.1 Compacting Good and Failing Test Sequences from *TG*

In the first case study, we performed our experiments with our clustering algorithm, while varying the overlap ratios between $0.1 \leq \tau \leq 0.9$ using 0.1 increments. The purpose is to determine the value of τ that achieves an optimal configuration of assertion clusters with a certain amount of similarity. From there, the clustered assertions are then used with our *TPO* and *PPR* techniques, which attempt to generate the least number of tests that satisfy (or falsify) the same set of assertions. We also generated test sequences without any compaction using *TG*. Our comparison is based on the number of test sequences generated from the three approaches (uncompacted and the two compaction algorithms) and the measured run-time in seconds. We used *MiniSAT* [10] as the SAT solver. These experiments were carried out on a Linux PC using a 2.4GHz 64-bit Dual-Core processor with 8GB RAM.

Table 4.1 lists the assertion benchmarks that were used in our case studies. The

Table 4.1: Assertion Benchmarks, Test Paths and Uncompacted Tests

Bench.	$\|A\|$	$\|\Pi_p\|$	$\|\Pi_f\|$	***TG***		***MyGen***	
				$\|T_p\|$	$\|T_f\|$	$\|T_p\|$	$\|T_f\|$
AXI	103	317	100	1080	221	374	100
FF-GC	37	93	41	145	70	61	41
FF-SC	80	159	80	238	159	192	156
PCI	39	586	373	13689	4639	4014	1320
SDRAM	30	472	70	20949	659	4980	569

first column (Bench.) is the name of the benchmark and the number of assertions ($|A|$) is shown in the second column. The third and fourth columns lists the number of test paths that were extracted from the acceptance (passing) ($|\Pi_g|$) and failure ($|\Pi_f|$) automata respectively. The number of tests generated from *TG* and *MyGen* are shown in columns 5-6 and 7-8 respectively, each consisting of the passing ($|T_p|$) and failing ($|T_f|$) test sets.

The set of extracted test paths were used by *TG* for generating the uncompacted tests whereas *MyGen* relies on a pseudorandom number generator for exerting the defined signals in the assertion. As seen in the table, the uncompacted test set size for *TG* is visibly larger than *MyGen*. *MyGen* considers only traversing through all the states within the automata, while *TG* uses a coverage-driven directed approach for generating tests that exercises all possible (and meaningful) sequences of events to either pass or fail the assertion.

Looking into the test set as a whole, the PCI and SDRAM assertion sets both generated a large number of good test sequences with a small number of assertions compared to other benchmarks. This is due to the fact that the PCI and SDRAM assertions contained temporal sequences with a large integer constant in the repetition operators causing both tools to generate many uncompacted tests. The number of failing tests for each benchmark is considerably less compared to its counterpart. This is due to the fact that the failing automata (discussed in Section 2.1.3) are a different representation of an assertion, which causes both tools to generate different test paths with Boolean events causing the assertion to fail. This table is used for comparing and calculating the reduced number of tests obtained using our two compaction approaches.

Table 4.2 shows the results obtained for compacting good test sequences using our two test compaction approaches with the four assertion clustering methods. (*Bench*) lists the name of the benchmark. The clustering modes are shown in the second

column (*Cluster*) using the abbreviations: Antecedent (*Ant.*), Consequent (*Con.*), Antecedent and Consequent (*Ant+Con*), Assertion Signal (*A-Sig.*). OTR portrays the *Overall Test Reduction* in terms of a percentage that is computed as:

$$OTR = \frac{|T| - |T^*|}{|T|} \times 100\% \tag{4.4}$$

where $|T^*|$ represents the number of compacted test sequences that were generated either from the *TPO* or *PPR* approaches, and $|T|$ represents the uncompacted tests that are listed in Table 4.1. Values in the fifth column are the overlap ratios (*Best* τ) which give an optimal configuration of clusters that achieves the least number of compacted tests. $|C|$ lists the number of clusters generated based on the chosen clustering mode and the overlap ratio. The run time (*Time*(s)) is the amount of time spent in the assertion clustering, test path generation and test compaction using *MiniSAT*. Values listed in the *Additional Test Reduction* (ATR) column represent the additional overall test compaction achieved when using our *PPR* test compaction mode. These values are computed as:

$$ATR = \frac{|T^*_{TPO}| - |T^*_{PPC}|}{|T^*_{TPO}|} \times 100\% \tag{4.5}$$

where $|T^*_{TPO}|$ and $|T^*_{PPC}|$ are the number of compacted tests using the *TPO* and *PPR* compaction strategies respectively.

Results obtained from both of our test compaction algorithms were able to achieve different amount of test reduction when using the different clustering modes. We observe that the run times using our two compaction schemes contributed very little performance overhead. We also observe that when using the *PPR* test compaction algorithm, at least 30% of additional test reduction (ATR) is obtained in all the assertion benchmarks.

The clustering modes and the overlap values (τ) produced different compacted sizes of tests for both compaction algorithms. In observing the number of tests generated by *TPO* and *PPR*, we see that *Antecedent* and *Assertion Signal* yielded the smallest test set size compared to other clustering modes. This is due to the fact that the assertions for each of the benchmarks contained more similar signals and sequences in the antecedent portion than in the consequent. Thus, using *Antecedent* and *Assertion Signal* clustering is beneficial for creating clusters of assertions with greater sharing of Boolean logic values with similar signals.

The overlap ratios (τ) vary significantly in each of the benchmarks and generate

Table 4.2: Comparison of Compaction and Clustering Modes for Good Test Sequences

Bench	**Cluster**	**Test Path Overlapping**					**Parallel-Path Removal**					
		$\|T^*\|$	OTR (%)	Best τ	$\|C\|$	Time (s)	$\|T^*\|$	OTR (%)	Best τ	$\|C\|$	Time (s)	ATR (%)
AXI	Ant.	271	75.1	0.3	9	0.57	**39**	96.4	0.3	9	0.53	**85.6**
	Con.	382	64.9	0.3	47	0.25	**227**	78.9	0.3	47	0.49	40.5
	Ant+Con	264	75.7	0.2	5	0.57	**43**	96.0	0.3	8	0.50	**83.7**
	A-Sig.	263	75.8	0.2	5	0.54	**32**	97.0	0.3	5	0.53	**87.8**
FF–GC	Ant.	47	67.6	0.5	7	0.02	**33**	77.2	0.3	4	0.09	29.7
	Con.	68	53.1	0.1	7	0.03	**43**	70.3	0.3	8	0.09	36.1
	Ant+Con	47	67.6	0.3	4	0.03	**32**	77.9	0.3	5	0.08	31.9
	A-Sig.	47	67.6	0.3	5	0.03	**32**	77.9	0.5	5	0.08	31.9
FF–SC	Ant.	82	65.5	0.5	4	0.19	**22**	90.7	0.4	4	0.25	**73.1**
	Con.	109	54.2	0.1	9	0.13	**45**	81.0	0.1	9	0.21	**58.7**
	Ant+Con	100	58.0	0.3	7	0.18	**27**	88.6	0.2	5	0.25	**73.0**
	A-Sig.	87	63.4	0.4	6	0.03	**30**	77.9	0.4	6	0.25	**65.5**
PCI	Ant.	589	95.7	0.2	2	1.68	**135**	99.0	0.2	2	2.34	**77.0**
	Con.	700	94.9	0.4	6	1.32	**255**	98.1	0.4	6	2.19	**63.5**
	Ant+Con	669	95.1	0.4	4	1.98	**213**	98.4	0.4	4	2.32	**68.1**
	A-Sig.	618	95.5	0.4	3	1.18	**148**	98.9	0.3	3	2.46	**76.0**
SDRAM	Ant.	945	95.5	0.9	2	3.37	**378**	98.1	0.9	2	4.97	**60.0**
	Con.	953	95.3	0.8	4	3.22	**402**	98.0	0.8	4	5.11	**57.8**
	Ant+Con	945	95.5	0.8	3	3.50	**402**	98.0	0.7	3	5.09	**57.4**
	A-Sig.	879	95.8	0.9	4	3.21	**414**	97.9	0.9	4	5.34	**52.9**

differing values of compacted sizes of tests. For instance, with the AXI and PCI benchmark, a low value of τ was required for achieving an overall test reduction of 75.8% and 95.7% respectively using *TPO* compaction. When using *PPR* compaction scheme, an additional reduction of tests was achieved, for an overall test reduction rate of 97% and 99% for the AXI and PCI assertion benchmarks, respectively. This is very favourable for reducing the overall verification time.

We note that the number of generated clusters produced from the different clustering modes varies moderately for each benchmark with the exception of the AXI assertions. When applying *Consequent* clustering on the AXI assertion set, five times more clusters were generated compared to *Antecedent* clustering and nine times more compared to the other modes. This shows evidence that the consequent portions of the AXI assertions had very minimal common signals.

For the FF-GC and FF-SC assertion sets, the overlapping ratios differ in all the clustering modes and test compaction methods; however, in most cases *Antecedent* clustering gave the most optimal set of tests using either *TPO* (OTR of 67.6% and 65.5% for FF-GC and FF-SC respectively) and *PPR* compaction (OTR of 77.2% and 90.7% for FF-GC and FF-SC respectively). This implies that the assertions for those benchmarks had more common signals and sequences in the antecedent portion compared to other portions of the assertion. Additionally, the *PPR* compaction algorithm achieved a further reduction of tests.

As for the SDRAM assertion set, it required a high overlapping ratio in all of the clustering modes in order to achieve a minimal test set. Thus, this goes to show that there were many common signals and sequences in both the antecedent and consequent parts of the assertion. For both compaction algorithms, *TPO* achieved 95.8% in overall test reduction whereas *PPR* had a further reduction of tests at 98.1%.

Table 4.3 shows the results for compacting failing test sequences by using all of our assertion clustering and our test compaction methods. Despite the difference in test sizes when compared to the good tests, it is still valuable to compact the failing test sequences in order to gain reductions in verification time. We see that our test compaction methodology was able to achieve an overall test reduction (using *TPO*) of at least 35% when applying *Antecedent* and *Assertion Signal* clustering in most benchmarks. For the AXI assertion set, it has garnered the least amount of compaction when using *Consequent* assertion clustering. As explained for compacting good test sequences, applying *Consequent* clustering on the AXI assertions generated many clusters due to very little similarity in the consequent portion of the assertion; however, when compared to the other clustering modes, they have produced

Table 4.3: Comparison of Compaction and Clustering Modes for Failing Test Sequences

Bench	**Cluster**	**Test Path Overlapping**					**Parallel-Path Removal**					
		$\|T^*\|$	OTR (%)	Best τ	$\|C\|$	Time (s)	$\|T^*\|$	OTR (%)	Best τ	$\|C\|$	Time (s)	ATR (%)
AXI	Ant.	129	41.6	0.4	12	0.24	**36**	83.7	0.4	12	0.38	**72.0**
	Con.	183	17.1	0.6	47	0.24	150	32.1	0.3	47	0.38	18.0
	Ant+Con	126	42.9	0.4	19	0.24	**32**	85.5	0.2	8	0.39	**74.6**
	A-Sig.	130	43.4	0.5	13	0.22	**36**	83.7	0.3	5	0.38	**72.3**
FF–GC	Ant.	42	40.0	0.6	6	0.03	26	62.8	0.6	6	0.07	38.0
	Con.	56	20.0	0.1	7	0.03	40	42.8	0.1	7	0.08	28.5
	Ant+Con	45	35.7	0.4	5	0.04	30	57.1	0.4	5	0.08	33.3
	A-Sig.	47	32.8	0.6	6	0.03	35	50.0	0.7	7	0.07	25.5
FF–SC	Ant.	90	43.3	0.6	10	0.15	**28**	82.3	0.5	4	0.21	**68.8**
	Con.	104	34.5	0.6	9	0.13	**45**	71.6	0.6	9	0.20	**56.7**
	Ant+Con	99	37.7	0.4	8	0.14	**30**	81.1	0.2	5	0.20	**69.6**
	A-Sig.	81	49.0	0.3	7	0.14	41	74.2	0.3	7	0.19	49.3
PCI	Ant.	2340	49.5	0.2	2	7.07	1556	66.3	0.2	2	5.41	33.5
	Con.	2351	49.3	0.6	8	5.19	1687	63.6	0.5	6	5.35	28.3
	Ant+Con	2373	48.8	0.4	4	6.12	1687	63.6	0.4	4	5.31	28.9
	A-Sig.	2334	49.6	0.4	3	5.18	1762	62.0	0.4	3	4.92	24.5
SDRAM	Ant.	445	32.4	0.9	2	0.25	266	59.6	0.9	2	0.27	40.2
	Con.	354	46.2	0.9	4	0.17	**173**	73.7	0.8	4	0.17	**51.1**
	Ant+Con	354	46.2	0.9	4	0.17	**173**	73.7	0.8	4	0.16	**51.1**
	A-Sig.	350	46.8	0.9	4	0.16	**170**	74.2	0.9	4	0.18	**51.4**

significantly fewer number of clusters and reduced number of tests.

In the PCI assertion benchmark, the number of compacted tests is considerably larger compared to the other benchmarks in this case study. This is due to the fact that the test paths generated from the failing automata have transition clauses requiring more than one Boolean logic value assignment, which prevents our compaction methods from effectively overlapping similar test paths and sharing similar logic values. Despite this drawback, our test compaction methodology was able to achieve 49.5% test reduction when using *Assertion Signal* clustering with the *TPO* compaction algorithm; however, a 66.3% reduction of tests was achieved with the *PPR* compaction algorithm when using *Antecedent* clustering.

In comparing both compaction modes and using the same assertion clustering techniques, we observe that the *PPR* compaction attained at least 51.1% of additional test reduction in most of the benchmarks. The *PPR* algorithm was particularly effective with the AXI assertion set. For instance, using *TPO* compaction and *Assertion Signal* clustering on the AXI benchmark yields an overall test reduction of 41.6%. When applying the *PPR* compaction algorithm with the same clustering mode, a further test reduction of 83.7% was achieved. The same can also be said for the FF-SC and SDRAM benchmarks. In the FF-SC assertion set, applying *Antecedent* clustering achieves a 43.3% of test reduction with *TPO* compaction while *PPR* compaction attains 82.3%. Similarly for the SDRAM assertion set, *TPO* compaction attains a 46.8% reduction of the test set when using *Assertion Signal* clustering whereas with *PPR* compaction attains 74.2%.

The results presented in this case study show that the overall effectiveness of assertion clustering and generating compacted test sets depend on how the assertions were written, specifically whether they contain many or minute similar signals and sequences. Our improved *PPR* compaction algorithm has produced an additional test reduction in all the benchmarks in both case studies. In the worst case scenario, for using the least effective assertion clustering mode (*Consequent* clustering for example), our test compaction methodology was able to reduce the sizes of test sets in half, for the passing and failing cases, when using the *PPR* algorithm. This can greatly reduce the amount of time spent in verification.

4.5.2 Compacting Good and Failing Test Sequences from *MyGen*

In the second case study, we applied our compaction to the test sequences obtained with the tool *MyGen* [2]. The passing and failing tests were compacted by our PPR algorithm. Every test is analyzed to determine the test paths that are *covered*, which are eventually removed from Π for further analysis. The objective is to demonstrate that our compaction methodology can be applied to other tools that are capable of generating tests from assertions. We applied the same clustering and overlap ratios listed in Tables 4.2 and 4.3 for the good and failing test sequences, respectively.

Table 4.4 lists the compacted tests for *MyGen*. The first two columns list the benchmark and the applied clustering mode. The number of compacted tests from *MyGen* is shown in the third column with the listed Overall Test Reduction (OTR) percentages in the fourth column. The fifth and sixth columns show the number of paths covered, and their percentage, respectively. The compacted failing tests are listed in the subsequent columns.

As seen in the table, our PPR algorithm was able to compact tests generated from *MyGen*. It should be noted that unlike our test generation, which has full coverage, the test sequences provided from *MyGen* do not attain 100% path coverage. The amount of path coverage determines how well the specification was explored and adequately exercised by the tests. For instance, a high OTR percentage was attained for the SDRAM and the PCI benchmarks when compacting the passing tests; however, those benchmarks only achieved 31.6% and 23.2% path coverage respectively. Similarly for the failing test sequences, the FF-SC attained the highest compaction compared to other benchmarks but was limited to 56% in path coverage. This is due to the fact that tests from *MyGen* did not necessarily assert the required signals for the generation of passing and failing test sequences, causing a sparse amount of test paths to transition the automata into their next states. The tool did not consider any kind of coverage where potential test paths are explored, thus not thoroughly exercising all the defined signals and sequences of an assertion. As for *TG*, our tool effectively explores the state space with more elaborate automata coverage metrics, which helps in including all transition clauses at least once.

We conclude that our compaction method can be successfully applied to other test generation approaches, but obviously cannot correct their coverage shortcomings.

Table 4.4: Compacting Passing and Failing Test Sequences of *MyGen*

Bench	Cluster	Passing Test Sequences				Failing Test Sequences			
		$\|T^*\|$	OTR (%)	Cov. Paths	% Cov. Paths	$\|T^*\|$	OTR %	Cov. Paths	% Cov. Paths
AXI	Ant.	174	53.5	205	54.8	57	43.0	55	54.5
	Con.	276	26.2	147	39.3	81	19.0	45	44.6
	Ant+Con	179	52.1	202	54.0	45	55.0	56	55.4
	A-Sig	174	53.5	205	54.8	45	55.0	56	55.4
FF-GC	Ant.	36	41.0	51	83.6	26	36.6	42	87.5
	Con.	39	36.1	50	82.0	21	48.8	40	83.3
	Ant+Con	36	41.0	51	83.6	22	46.3	38	79.2
	A-Sig	33	45.9	52	85.2	26	36.6	42	87.5
FF-SC	Ant.	99	48.4	112	70.4	28	82.1	38	47.5
	Con.	91	52.6	143	89.9	34	78.2	42	52.5
	Ant+Con	89	53.6	124	78.0	30	80.8	45	56.3
	A-Sig	82	57.3	116	73.0	35	77.6	26	32.5
PCI	Ant.	382	90.5	86	14.7	438	66.8	31	8.3
	Con.	450	88.8	136	23.2	426	67.7	61	16.4
	Ant+Con	258	93.6	111	18.9	397	69.9	26	7.0
	A-Sig	251	93.7	101	17.2	397	69.9	26	7.0
SDRAM	Ant.	470	90.6	146	30.9	410	27.9	22	31.4
	Con.	455	90.9	148	31.4	338	40.6	7	10.0
	Ant+Con	420	91.6	149	31.6	338	40.6	7	10.0
	A-Sig	420	91.6	149	31.6	338	40.6	7	10.0

4.6 Summary

Directed test generation from assertions can produce a large volume of tests for use in design verification. This chapter has presented a method for compacting directed tests based on assertions. One of the contributions that was presented in this chapter is the development of a method that groups similar assertions together to generate a compact set of good and failing test sequences. Assertion grouping is based on the antecedent, consequent, combined antecedent and consequent, and assertion signal clustering approaches. Results show that the proper choice of clustering method and its main parameter (the overlap ratio) can be used to obtain even more test compaction. To this effect, we also developed two efficient compacted test generation algorithms, namely *Test Path Overlapping* (*TPO*) and *Parallel-Path Removal* (*PPR*), which were used to compact good and failing test sequences for a set of assertions.

The experimental results show that the test path overlapping method can attain at least 50% and 30% reduction for the good and failing test sequences, respectively; the parallel path elimination approach reduces the test set size by at least 70% and 50%, for the good and failing cases respectively. Results show that both of the algorithms have compacted tests on a set of 289 assertions, with very little performance overhead. We also show that our tool can be interfaced with the most related test generation developments in our area, namely the *MyGen* tool, for compacting test sets from other sources. The significant compaction improvement provided by the *PPR* algorithm is favourable for reducing the overall verification time. Efficient test generation is essential given the increased complexity faced by modern verification endeavours, in the quest for bug-free designs.

Chapter 5

Efficient Data Encoding of Mutation and Fault Data on GPUs

The obstacle of using Graphics Processing Units (as described from Section 2.3) for accelerating digital circuit simulation is to devise a data-parallel representation of the circuit data. Electronic Design Algorithms, particularly logic and fault simulation, require a certain level of data dependency, which can pose a challenge in generating data-parallel representations of a circuit.

This chapter presents two proposed tools for mutation and fault simulation using GPUs, namely *μ-GSIM* and *GS-SIM* respectively. These tools rely on a novel data-parallel generation algorithm, which exploits several levels of parallelism that is suitable for simulation on GPUs. Two proposed encoding techniques were also developed for storing gate, mutant and fault data using the GPU's computer word. These proposed approaches have helped in reducing the memory consumption, which ultimately led an improvement in simulation performance on the GPU. The goal of this chapter is to establish an accelerated framework, which will be used for simulating mutated designs in the context of design verification.

5.1 Motivation

Graphics Processing Units (GPUs) have gained wide spread popularity as being a low cost parallel platform for accelerating scientific computations. As Advanced Programming Interfaces (APIs) for these coprocessors have improved, many different forms of computation are being offloaded from the host CPU, with significant speed-ups being achieved [27]. Recently, GPUs have been used for accelerating compute intensive applications in the field of Electronic Design Automation (EDA) [106]. Tasks that can

benefit from this advanced parallel platform are logic and fault simulation algorithms, which can be used for accelerating the simulation of large mutated circuits for evaluating the test quality. In order to take advantage of the wealth of processing power provided by GPUs, a few key requirements need to be met, such as ensuring as much data independency as possible and properly organizing the data in the GPU's device memory. GPUs employ the Single Instruction Multiple Data (SIMD) paradigm so that the many threads deployed on the GPU are executing an instance of the kernel program while operating on different data.

Mutation testing and fault simulation can benefit from the massive parallelism provided by GPUs. For instance, given a Design Under Test (DUT) with a large test set, the design is simulated for a set of mutations or stuck-at faults, in which the goal is to observe the differences in the outputs compared to the error free circuit. Since the mutations and stuck-at faults are independent of each other, they can be simulated in parallel by creating multiple instances of the circuit, each containing a different injected error. Additionally, further parallelism can be exploited by having each thread on the GPU simulate a circuit containing a set of mutants (or stuck-at faults) that is stored over the length of the GPU's computer word, thus leveraging the inherent bit parallelism of GPUs. We attempt to exploit many levels of parallelism within a single thread, which requires an efficient mapping and duplication of the circuit, mutant and fault data.

In this chapter, we present a novel GPU-based approach for accelerating simulations of digital circuits for mutation testing and fault simulation. Our aim is to generate an efficient data mapping of multiple circuit designs in order to achieve optimal performance of logic-based simulations on GPUs. Logic-based simulation requires a certain level of data dependency, which poses a challenging task for creating an efficient mapping of the gate, mutant and stuck-at fault data. We show that our optimized mapping of a plethora of circuit designs (injected with either many mutations or faults) on the GPU and has contributed significant performance improvements compared to when naively duplicating a circuit for every injected mutant or stuck-at fault.

The contributions brought forth into this chapter are summarized as follows: [1]

- We developed a logic-based circuit simulator that exploits the inherent bit par-

[1] The contents of this chapter is based on the papers entitled *Mu-GSIM: A Mutation Testing Simulator on GPUs* [14] and *Efficient Data Encoding for Improving Fault Simulation Performance on GPUs* [13]

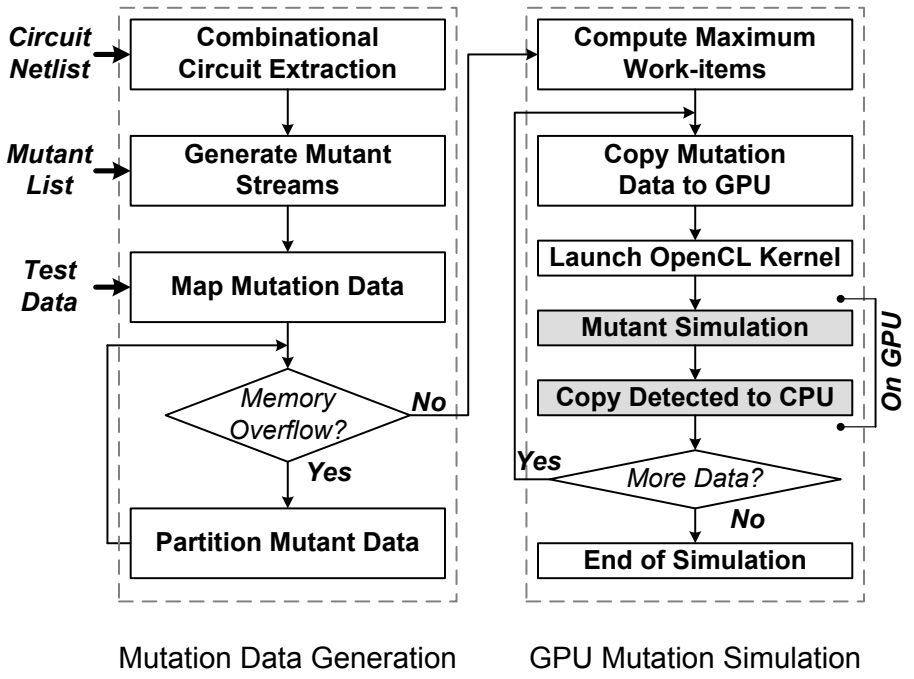


Figure 5.1: μ-*GSIM* Framework

allelism of GPUs for improving mutation testing and fault simulation applications;

- We proposed two different encoding schemes that enables efficient mapping of circuit data on the GPU that leads to improved simulation performance.

The contents of this chapter is organized as follows: Sections 5.2 presents the proposed mutation-based simulator, μ-*GSIM* where the details of the algorithms are given in Sections 5.2.1 through 5.2.2. The results obtained from μ-*GSIM* are presented in Section 5.2.5, where comparisons are made with the different mutant encoding techniques and a commercial gate-level event-driven simulation tool. Following, is a presentation of the *GS-SIM* where we use the similar memory encoding techniques for developing the fault simulation tool on GPUs. Section 5.3 presents the overview of *GS-SIM* where the encoding techniques and the GPU fault simulation algorithm is presented in Sections 5.3.1 and 5.3.2 respectively. Results from *GS-SIM* are shown in Section 5.3.4.

5.2 μ-*GSIM* Overview

Figure 5.1 shows the overall μ-*GSIM* simulator framework. Our approach leverages the inherent bit parallelism of GPUs where we use the entire length of the GPU's com-

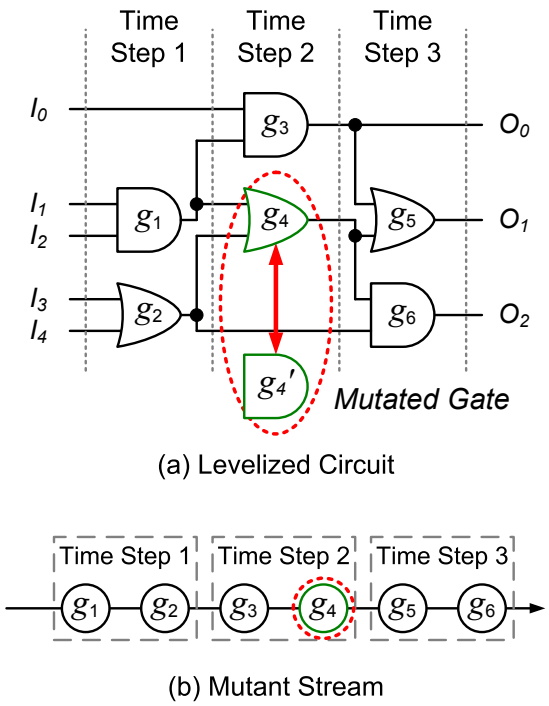


Figure 5.2: Circuit to Mutant Array Transformation

puter word for representing the combined gate-mutant data and mutant detectability data. *μ-GSIM* consists of both sequential (non-shaded boxes) and data-parallel (shaded boxes) tasks that are performed on the host (CPU) and GPU respectively. Our approach takes as input the circuit netlist and a list of mutants that describes the inserted erroneous logic gate or fault and the associated test data. The flow of our tool is divided into two phases. The first phase, namely *Mutation Data Generation*, is responsible for generating the gate-mutant data and creating an effective memory mapping so that all work-items on the GPU can operate independently. In the event of a memory overflow, the host program will partition the mapped mutation data and then will perform an additional kernel execution. The second phase is *GPU Mutation Simulation*, where the host determines the effective work-item and work-group configuration in order to achieve optimal performance when performing logic simulation on a circuit with injected mutants. In the next subsequent sections, we thoroughly describe these phases of our tool in detail.

5.2.1 Mutant Stream Generation

This section describes the tasks involved for creating an efficient data parallel representation of a circuit containing mutated logic gates. To perform logic simulation on a

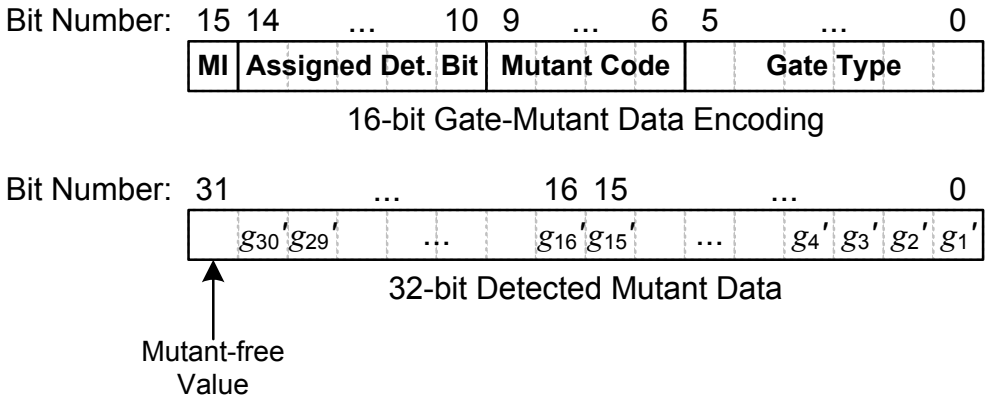


Figure 5.3: Multiple Mutant Gate (MMG) Encoding and Detectability Word

mutated circuit, *μ-GSIM* uses the concept of *mutant streams* so that every work-item simulates the same set of gates containing different mutants. We define a *mutant stream* as m_i, which represents a one dimensional array of logic gates of a circuit (C). The gates within the stream are arranged in a particular order so that their input and output dependencies are satisfied. A *mutated gate*, defined as g', is a gate depicting the intended error such that their gate types are not the same ($g_i \neq g_i' \mid g_i \in C$). The erroneous gate replacements that we use are defined in [84].

Figure 5.2(a) illustrates a levelized circuit netlist containing combinational logic elements. We see that the circuit contains a mutated gate, labelled as g_4', that replaces the *or*-gate with an *and*-gate. After applying the *As Soon As Possible* scheduling algorithm to the circuit, the set of gates from each time step gets transformed into an array as shown in Figure 5.2(b). Each labelled node depicts a specific gate inside the circuit. Nodes that are outlined in green depict the mutations that were added, as is shown at node 4. Every work-item will evaluate the same gate type in a lockstep manner. This implies that every work-item will follow the same execution path, thereby avoiding branch divergence.

The set of gates within a mutant stream are represented by a 16-bit data word that is shown in Figure 5.3. This *Multiple Mutant Gate* (MMG) encoding scheme allows for every work-item to perform simulation on a set of gates with multiple injected mutants independently. The first 6 bits (bits 0 through 5) of the word are used to store the type of logic gate. A control flag called *Mutant Injection* (MI) is used for representing every mutated gate, g_j'. This flag is used by the simulation kernel for computing and placing the incorrect result over the 32-bit word. A *Mutant Code* represents a particular gate type substitution that depicts the intended incorrect gate

Algorithm 5.1 Mutant Stream Generation

FUNCTION: *mutant_stream_gen*
Input: Mutant List (G'_{list}), Circuit (C), Gate Library (G_{lib})
Output: Set of Mutant Streams (MS)

$i \leftarrow 0$
while G'_{list} != $\emptyset$ **do**
  Initialize m_i with encoded gate types, $det\_bit \leftarrow 0$
  for all mutant $g'_l \in G'_{list}$ **do**
    if $|G'_i| < 31$ **then**
      $k \leftarrow$ array position of $g_l \in m_i$ for injecting g'_l
      $MI \leftarrow 1$, $mu\_code \leftarrow g'_l$, $gate\_type \leftarrow G_{lib}(g_l)$

      /* Encode 16-bit word */
      $m_i[k] \leftarrow (MI|det\_bit|mu\_code|gate\_type)$
      $det\_bit \leftarrow det\_bit + 1$
    if $|G'_i| > 31$ **then**
      Break loop and start new mutant stream

  Remove added mutants from G'_{list}
  $MS \leftarrow MS \cup m_i$, $i \leftarrow i + 1$
return (MS)

that is injected into the circuit. The *Assigned Detection Bit* (shown as *Assigned Det. Bit* from the figure) specifies the number of left shifts for appropriately inserting the incorrect result over the 32-bit word. This value can range from 0 to 30, while leaving bit 31 as the mutant-free value. The true value is used for storing the correct simulation results and also for computing the number of killed mutants. We note that the gate-mutant encoding scheme can be expanded over a 32-bit word if the number of gate or mutant types exceed 63 and 15 respectively.

Algorithm 5.1 describes the mutant stream generation approach which transforms the circuit to the representation as illustrated in Figure 5.2(b). The inputs to the algorithm are a levelized circuit netlist (C) and a list of mutants (G'_{list}). The gate library (G_{lib}) is a lookup table function which returns a 6-bit code depicting the gate type. Every mutant ($g'_l \in G'_{list}$) specifies a logic gate in the circuit (C) that is to be replaced by the erroneous gate type, which is represented by a mutant code (mu_code).

The algorithm begins by creating a new mutant stream m_i where every logic gate is already encoded with its gate type. Each mutant g'_l from the list is assigned to a mutant stream m_i if the size of the set of mutants for the current stream $|G'_i|$ is

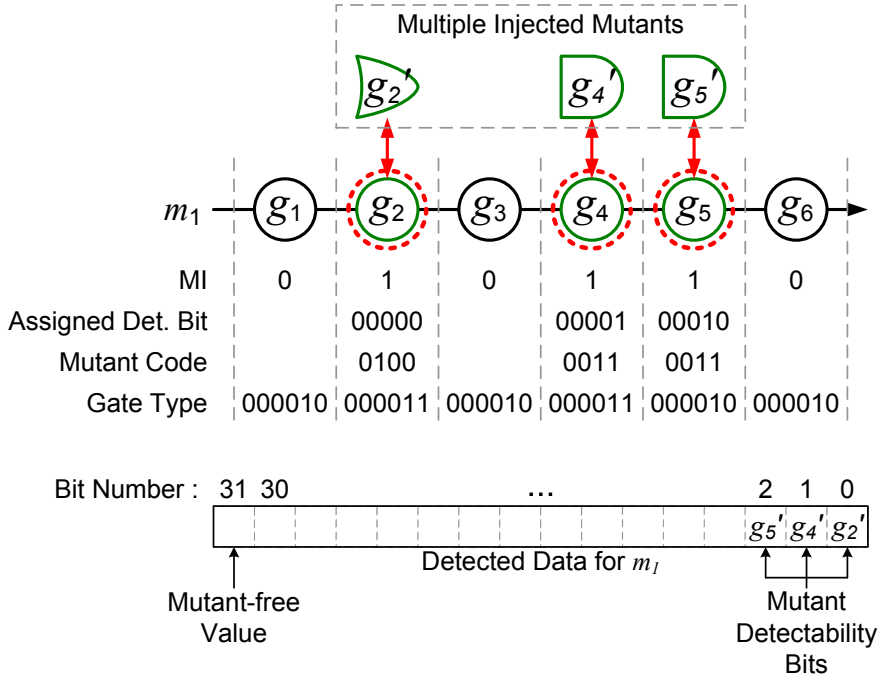


Figure 5.4: Mutant Encoding Example

less than 31. Once this condition is satisfied, the injected mutant is placed at the k^{th} position in the array of logic gates of m_i. The value of k is determined by the specified mutated gate g_l'. Then, the control bit MI is set to logic-1 to indicate that the k^{th} logic gate is a mutant defined by the code, *mu_code*. The value of the assigned detection bit is determined by the counter *det_bit*, which represents the current unassigned bit position used for detecting the killed mutant of g_l'. The variable *gate_type* holds the 6 bit code for representing the gate type. After the data have been retrieved, the algorithm combines the necessary data portions into a 16-bit word and stores it into the k^{th} gate of the mutant stream ($m_i[k]$). Thereafter, the subsequent mutant g_{l+1}' will be assigned to the next unassigned bit until $|G_i'|$ exceeds 31 or all the mutants from the list (G_{list}') are assigned.

Figure 5.4 shows an example of a mutant stream m_1 containing multiple mutants that are encoded over the 16-bit data word. Each data field is split into rows for illustration purposes. We see that the mutant stream contains three injected mutants, namely g_2', g_4' and g_5'. Those gates will have MI set to logic-1, depicting that the gate has been replaced by a specific erroneous gate defined in the mutant code, *mu_code*. Mutant-free gates will have MI set to logic-0 so that each work-item will compute the gate's output normally. The assigned detection bit holds the value of the bit position that will be used for detecting the particular mutant. Also shown in the figure is the

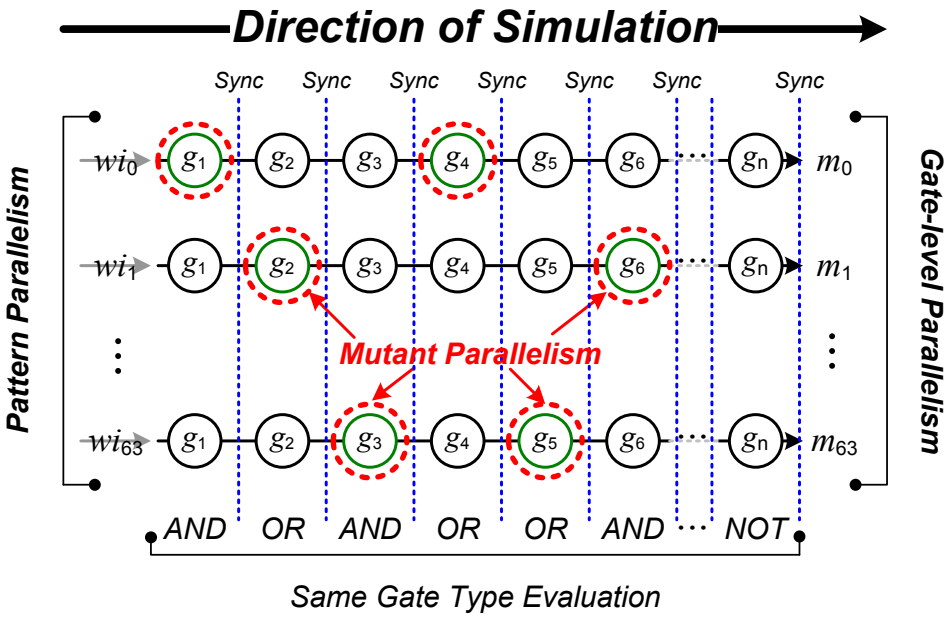


Figure 5.5: Exploited Parallelism Factors

32-bit detected data word which explicitly assigns a detected bit for every mutated gate, g'_l. Whenever the assigned detected bit is at logic-1, then the mutant at that bit location is deemed killed, otherwise it was not detected.

The mutant stream approach and the 16-bit encoding scheme presented in this section will now be used for evaluating the mutated circuit on the GPU. The next section discusses the second phase of the μ-*GSIM* tool.

5.2.2 GPU Mutation Simulation

In this section, we describe our proposed GPU circuit simulator for mutated circuits.

5.2.3 Simulation Kernel

The principle operation of a single work-item on the GPU is to perform logic simulation of a mutated circuit by leveraging several parallelism factors that were encoded within a mutant stream, m_i. Figure 5.5 outlines the different levels of parallelisms that were achieved. First, we have exploited *gate-level parallelism* by having a group of 64 work-items simulating a set of 64 mutant streams. Every work-item evaluates the *same non-mutated gate type* and synchronizes at the end of each evaluation, which ensures that no branch divergence can occur. Second, *mutant parallelism* was exploited because every mutant stream contains a unique set of replaced gates or

Algorithm 5.2 Mutant Stream Simulation Kernel

FUNCTION: mutant_stream_sim
Input: Mutant Stream Set (MS), Intermediate values ($nets$), Detected array ($detected$)

/* Initialize Variables*/
$num_{wi} \leftarrow$ **get_global_size**(0) // Total Num Work-items
$wi_{id} \leftarrow$ **get_global_id**(0) // Work-item ID
$det\_local \leftarrow 0x00000000$ // Detected data
$m_i \leftarrow MS[wi_{id}]$ // Mutant Stream

for all logic gates $g_j \in m_i$ **do**
  /* Decode Mutant and Gate Data */
  $mg\_data \leftarrow m_i[j \times num_{wi} + wi_{id}]$
  $MI \leftarrow mg\_data$ & 0x8000
  $mu\_code \leftarrow$ (RHS 6 bits of mg_data) & 0x000F
  $det\_bit \leftarrow$ (RHS 10 bits of mg_data) & 0x001F
  $gate\_type \leftarrow$ & 0x003F

  /* Pass decoded data to mutant_sim kernel */
  mutant_sim ($gate\_type$, MI, mu_code, det_bit, $nets$)

/* Compute Detected Data */
for all output ports p of $MS[wi_{id}]$ **do**
  $cor \leftarrow net_{31}$ & 0xFFFFFFFF | $(1 - net_{31})$ & 0x00000000
  $det\_local \mid= (nets[p \times num_{wi} + wi_{id}] \oplus cor)$

/* Store Detected Data to Device Memory */
$detected[wi_{id}] \leftarrow det\_local$

errors that were injected into the circuit and can be simulated independently. Third, *pattern parallelism* was fulfilled by having every work-item simulating the same set of tests that are applied to the primary inputs of the circuit. These levels of parallelism were encoded over the 16-bit data representation, which enables a group of 64 work-items to simulate 1984 different mutants (31 mutants per mutant stream × 64 mutant streams) concurrently.

Algorithm 5.2 describes our proposed logic simulation kernel that operates on a mutant stream. This kernel code is executed by every work-item on the GPU. The inputs to the kernel function are the mapped mutant stream data MS, an array called *nets* that is used for storing the primary input, intermediate and primary output values during logic simulation and an array *detected* for storing the detected

mutant data. The input data are accessed through the GPU's device memory in a coalesced fashion.

The algorithm begins by retrieving the appropriate offset values, namely the total number of work-items deployed and the work-item ID, which are accessible through the built-in OpenCL functions `get_global_size(0)` and `get_global_id(0)` respectively. The index `(0)` returns the one dimensional ID value of the work-item. These offset values are important for computing the effective addresses so that every work-item accesses the GPU's device memory in a coalesced fashion. The detected variable, *det_local*, is used for storing the 32-bit detected data type in local memory. We use local memory because *det_local* is continuously accessed and updated at every completion of a simulation run for a set of tests. This reduces the additional latency when accessing the GPU's device memory. The mutant stream m_i is then fetched by accessing the global memory variable MS and using the work-item's global identifier as the index. We also note that every work-item will initialize the *nets* memory based on the test data used for verifying the circuit.

The simulation kernel continues by decoding the 16-bit data type that is stored in *mg_data*, which is done by applying the appropriate logic operations and bit shifts as indicated in the algorithm. This is for retrieving the gate type (*gate_type*), mutation injection control flag (MI), the assigned detection bit (*det_bit*) and the mutant code (*mu_code*). The decoded data are then sent to a function called `mutant_sim` which is used to determine which gate evaluation function is called based on the coded gate type. After all the gates are evaluated in the mutant stream m_i, the algorithm proceeds by computing the number of detected mutants. The mutant-free value is obtained by retrieving the most significant bit (bit 31) from *nets* which gets expanded over the 32-bit word and stored into the variable *cor*. Then, a logical *xor* operation is performed on the intermediate values that depict the primary outputs. Finally, the detected word stored in local memory is transferred to the GPU's device memory in a coalesced fashion.

The function `mutant_sim` was called from Algorithm 5.2 which is used for invoking the appropriate gate evaluation function based on the decoded value of *gate_type*. A gate evaluation function is defined for every type that is supported in the gate library. For instance, the 6-bit gate type code `000010` represents a two input *and*-gate. The `mutant_sim` function calls `and2_mutant_sim`, which is shown in Algorithm 5.3. Evaluation functions for other gate types are similar to the one presented.

The primary operation of every gate evaluation function is to compute the mutant-free and the mutated value of a logic gate. At the beginning of the evaluation function,

Algorithm 5.3 Mutant Gate Function for Two Input *and*-gate

FUNCTION: and2_mutant_sim
Input: Input net addresses ($in0$, $in1$), Output net address (out), $nets$, MI, mu_code, det_bit

/* Retrieve Input Values */
$in0\_v \leftarrow nets[num_{wi} \times in0 + wi_{id}]$
$in1\_v \leftarrow nets[num_{wi} \times in1 + wi_{id}]$

/* Compute correct value */
$m\_free \leftarrow in0\_v \;\&\; in1\_v$

/* Compute mutated values */
$m\_value \leftarrow$
$(mu\_code) \times (!(in0\_v \;\&\; in1\_v) \mid$ // mutant-*nand*
$(mu\_code) \times (!(in0\_v \mid in1\_v)) \mid$ // mutant-*nor*
$(mu\_code) \times (in0\_v \;\&\; in1\_v) \mid$ // mutant-*and*
$(mu\_code) \times (in0\_v \mid in1\_v)$ // mutant-*or*

/* Write Result to $nets$ */
$nets[num_{wi} \times out + wi_{id}] \leftarrow$
(1-MI) & $(m\_free) \mid (MI)$ & (LHS m_value by det_bit)

the intermediate values are retrieved and stored into the 32 bit variables $in0\_v$ and $in1\_v$. Then, the mutant-free value is computed, followed by the computation of the mutated values. The bits of each mutant code mu_code will choose which incorrect result to store into m_value. The gate evaluation algorithm shows only a subset of the mutant functions due to space reasons. Finally, a Left Hand Shift (LHS) operation on m_value is performed for inserting the erroneous output at the appropriate bit position defined by det_bit. This ensures that the intermediate values for detecting other mutants in m_i are not overwritten. Injecting the incorrect result into the intermediate values ($nets$) depends on the value of the control flag MI. If MI is set to logic-1, the mutated value is inserted into the intermediate values stored in $nets$, otherwise the mutant-free value is used.

5.2.4 Maximum Work-item Configuration and Memory Scalability

Determining the maximum number of work-items to deploy on the GPU depends on the available memory remaining after mapping the mutant stream data. The host

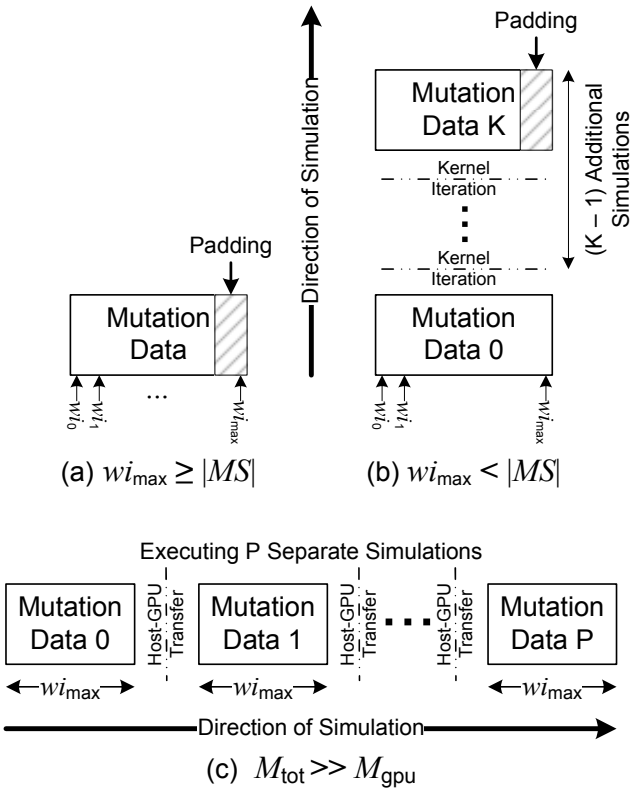


Figure 5.6: Different Mapping and Simulation Scenarios

program computes the maximal number of work-items, defined as $wi_{\max}$, which is dependent on the amount of memory in bytes that is consumed by the set of mutant streams M_{MS} and the number of bytes for storing the intermediate values M_{int} per work-item. We define M_{tot} as the total memory that is required for simulating the entire set of mutant streams and it is computed by:

$$M_{\text{tot}} = M_{\text{MS}} + M_{\text{nets}} \tag{5.1}$$

The memory required for storing the mutant stream data, M_{MS} is defined by:

$$M_{\text{MS}} = (|m_i| \times |MS|) \times \text{sizeof}(mg\_data) \tag{5.2}$$

where $|m_i|$ represents the length of the mutant stream in terms of the number of logic gates.

Initially, we compute the raw number of work-items that can potentially be instantiated as the following:

$$wi_{\text{raw}} = \frac{M_{\text{rem}}}{M_{\text{int}}}, \quad \text{with} \quad M_{\text{rem}} = M_{\text{gpu}} - M_{\text{MS}} \tag{5.3}$$

where M_{int} is the number of bytes used for storing the intermediate values in every gate stream; however, we require that the number of work-items be a factor of 1024 so that we can have at least 128 work-items in each Graphics Processing Cluster (1024 work-items / 8 Graphics Processing Clusters). Thus, from wi_{raw} we derive the maximum number of work-items with:

$$wi_{\text{max}} = 1024 \times n, \quad \text{with} \quad n = \lfloor wi_{\text{raw}}/1024 \rfloor \tag{5.4}$$

We chose the floor operator when computing n so that the memory allocated for M_{nets} along with the mutant stream data, M_{MS}, does not exceed the memory available provided by the GPU (M_{gpu}).

Computing the maximum number of work-items is limited to the memory consumed by the mutant stream data and the memory required for storing the intermediate values. Additionally, the number of work-items deployed can affect the mapping of the data along with the overall performance of the simulation kernel. Figure 5.6 shows the different data mappings and the flow of simulation. The host program aims to allocate as many work-items as possible in order to simulate the entire set of mutant streams without requiring additional simulation (iteration) runs. This occurs when $wi_{\text{max}} \geq |MS|$, which is shown in Figure 5.6(a) whereby each work-item is simulating a mutant stream. Padding of the memory is needed in order to align the mutation data to the number of work-items deployed.

When the value of wi_{max} is less than the size of MS, the host program creates a mapping of the mutation data of the form that is shown in Figure 5.6(b). This occurs when the mutation data consumes a significant amount of memory preventing the host program from allocating sufficient work-items. The kernel will then evaluate $K - 1$ additional simulations, where K is defined as:

$$K = \left\lceil \frac{|MS|}{wi_{\text{max}}} \right\rceil \tag{5.5}$$

The additional kernel runs will add to the overall simulation time.

Figure 5.6(c) shows when the total memory M_{tot} required for simulation greatly exceeds the memory capacity on the GPU ($M_{\text{tot}} \gg M_{\text{gpu}}$). This causes our host program to partition the set of mutant streams into P equal sets where each subset of mutant streams is simulated separately; however, this requires more than one data transfer from the host to the GPU. Thus, it is essential that the number of partitioned mutant streams be kept low.

5.2.5 Experimental Results for μ-*GSIM*

In this section, we evaluated the performance of our μ-*GSIM* tool against a commercial event-driven gate-level simulator tool on a set of large circuits from the ITC'99 benchmark suite [107]. Due to the commercial tool's licensing agreement, the name cannot be mentioned in this thesis. Smaller ITC'99 benchmarks take negligible time with our tool and are not reported due to space reasons. Our experimental platform is based on a six-core AMD processor running at 3.2 GHz with 16 GB of memory, which serves as the host. The GPU device is a GTX 560 Ti graphics card containing 8 GPCs, each consisting of 48 processing cores with a core clock speed of 1.7 GHz. The available on-board memory is 1 GB with a maximum bandwidth of 128 GB/s. We used the NVIDIA CUDA SDK 4.2.1 using version 1.1 of the OpenCL language.

To achieve the best performance, we aim to generate an efficient data mapping so that the GPU can concurrently simulate as many mutant streams as possible. This implies that the work-items (one work-item per mutant stream) should efficiently use the available resources provided by every GPC. In our GPU platform, each GPC consists of 48 KB of local memory and 32,768 32-bit registers which are shared by every work-item [30]. The maximum number of concurrent work-items that can occupy a GPC is 1536, which limits each work-item to 32 bytes of local memory and 21 registers. Our implemented OpenCL kernel consumes 20 registers and 4 bytes of local memory, thereby our approach is capable of simulating 1536 mutant streams concurrently within a GPC and a total of 12228 mutant streams when utilizing all the GPCs of the GPU.

Prior to executing our simulation kernel onto the GPU, we first compute and compare the maximum number of work-items ($wi_{\max}$) when using two different mutant encoding schemes, namely *Single Mutant Gate* (SMG) and Multiple Mutant Gates (MMG). The SMG encoding injects one mutated gate whereas MMG encoding attempts to inject multiple mutants into a mutant stream. The purpose is to show the effectiveness of our multiple mutant gate encoding scheme in the attempt to optimize the memory usage on the GPU. The computed values of $wi_{\max}$ for each encoding technique are then used for launching our OpenCL kernel on the GPU where we compare the performance run-time between the two encoding schemes along with the run-times of the commercial tool executing on the host. Each mutation is assumed to be a gate replacement error as described in [84]. A copy of a circuit is generated for every mutation and we cumulatively injected mutations in increments of 1000, such that the first 1000 mutants in the case of 2000 injected mutations are identical to those in the case of 1000 mutations, and so forth. Each benchmark was simulated

Table 5.1: Memory Usage Comparison and Maximum Work-item Computation for 5000 Injected Mutations

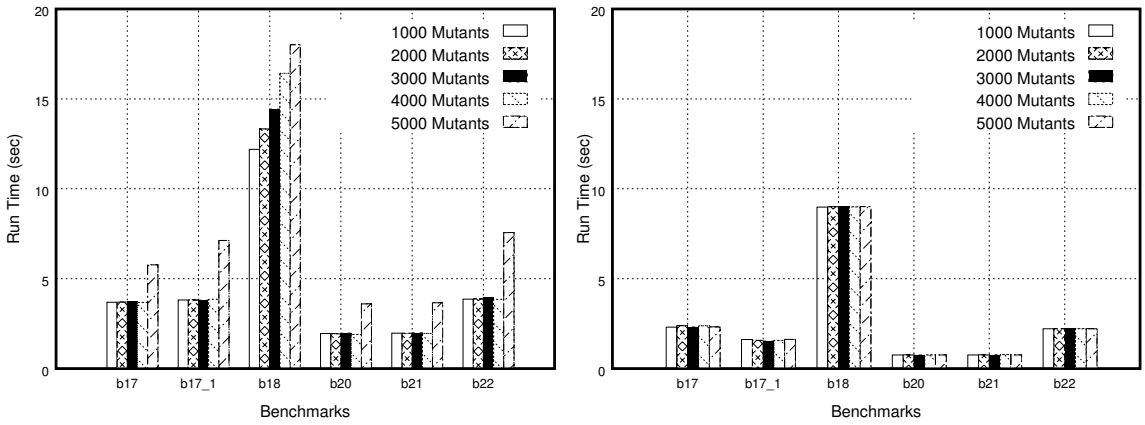
Circuit	**Gates**	**Encoding**	M_{MS} (MB)	M_{nets} (MB)	M_{tot} (MB)	wi_{max}
b17	30777	SMG	480.9	503.6	984.5	4096
		MMG	300.6	629.5	930.0	**5120**
b17_1	38166	SMG	447.3	309.9	757.2	4096
		MMG	298.2	619.8	918.0	**5120**
b18	111241	SMG	2172.7	895.5	3068.2	2048
		MMG	434.5	895.5	1330.0	2048
b20	19682	SMG	461.3	315.7	777.0	4096
		MMG	307.5	631.4	938.9	**8192**
b21	20027	SMG	469.4	321.1	790.5	4096
		MMG	312.9	642.2	955.1	**8192**
b22	29162	SMG	455.7	467.6	923.3	4096
		MMG	284.8	584.6	869.3	**5120**

using 131072 (128K) tests for ten runs, after which the run-times are averaged in order to ensure consistent measurements of the execution times.

5.2.5.1 Computing Maximum Work-item Configuration

Table 5.1 shows the memory usage and the computed maximum number of work-items that can be launched onto the GPU. The first column (Circuit) shows the circuit benchmark along with the number of logic gates (Gates) listed in the second column. The two mutant encoding schemes that were used are shown in the third column (Encoding) where we compare the memory consumption and computed maximum work-items by inserting a single mutated and multiple mutated gates into a mutant stream. The fourth and fifth columns list the memory consumed by the mutated circuits (mutant streams) (M_{MS}) and the storing of the intermediate values (M_{nets}) by every work-item respectively. The total memory shown in the sixth column (M_{tot}) is the sum of M_{MS} and M_{nets}. The computed maximum number of work-items (wi_{max}) is shown in the seventh column and is based on the approach described in Section 5.2.4. We set the value of M_{gpu} at 960MB so that the GPU can use the remaining memory for storing software drivers used for communication with the host. The memory and work-item values listed in the table are based on injecting 5000 mutations.

As seen in the table, the memory consumed by the mutated circuits (M_{MS}) for each benchmark is larger when only one mutant is injected in each mutant stream. When applying our multiple mutant encoding scheme, the memory consumption is



(a) Single Mutant Gate Encoding

(b) Multiple Mutant Gate Encoding

Figure 5.7: Run-times of *μ-GSIM* for Different Mutant Encoding Techniques

reduced by 60% for circuits `b17` and `b22`. The largest decrease in memory consumption was observed in `b18` where we reduced the memory foot print for storing the mutated circuits by 400% ($\frac{2172.7-434.5}{434.5}$). This is due to the fact that we were able to assign multiple mutants within a mutant stream and have its detectability data assigned over the 32-bit computer word.

Most benchmarks experienced an increase in the number of work-items ($wi_{\max}$) that can be launched onto the GPU. Reducing the memory usage for storing the mutated circuits (M_{MS}) has allowed our algorithm to allocate more work-items, thereby allowing more mutant streams to be simulated concurrently. For instance, circuits `b17_1`, `b20` and `b21` observed an increase in the number of work-items by as much as 2.5× compared to when using single mutant encoding. This goes to show that memory efficiency is key for potentially increasing the performance for applications running on the GPU.

Circuit `b18` did not have any increase in the number of work-items. Due to its large size, the memory consumption for storing the intermediate values exceeds the available capacity of the GPU's device memory, M_{gpu}. Decreasing the number of work-items below 1024 would not provide any performance gains as not all resources of the GPC are effectively utilized. We should note that the total memory consumption (M_{tot}) has decreased significantly by 150% when using our multiple mutant encoding scheme, which helps to reduce the amount of CPU to GPU transfers required during simulation.

5.2.5.2 Run-time Comparison with Different Mutant Encodings

Figure 5.7 shows the run-time comparisons when employing the SMG and MMG mutant encoding schemes. Each measured run-time for every benchmark was based on the number of injected mutations as indicated in the graphs. We see that the run-times are reduced when employing the MMG encoding technique, which was able to achieve a speed-up of more than 2.5× compared to the SMG encoding. Additionally, the allocation of more work-items onto the GPU also help with masking the high latencies involved in accessing the GPU's device memory.

We also observe that when increasing the number of mutations, the run-times for every benchmark remain consistent for the MMG encoding technique. This is due to the fact that the number of work-items that were launched onto the GPU is sufficient for simulating the entire set of mutant streams concurrently. For the SMG encoding, the run-times increase when injecting 5000 mutations into the circuit. This occurs because the computed maximum number of work-items ($wi_{\max}$) was less than the size of the mutant stream set ($wi_{\max} < |MS|$). Thus, reducing the amount of concurrent mutant streams that are being simulated and requiring an additional execution of the simulation kernel.

For circuit `b18`, the SMG run-times increase with each successive number of injected mutations. This is caused by the additional data communication that is required between the host and the GPU since the mapped data (M_{tot}) exceeds the GPU's device memory capacity. Using the MMG encoding allows the benchmark to reduce its memory consumption for storing the mutated circuit data and requiring the same number of data memory transfers between the host and the GPU. This goes to show that efficient memory utilization is key for improving the overall performance of the simulation kernel.

5.2.5.3 Run-time Comparison Against Commercial Tool

Table 5.2 compares the run-times of μ-*GSIM* with a commercial tool (shown as Comm. Tool) that was used for simulating different sets of mutated circuits, whereas Table 5.3 shows the attained speed-ups. A direct comparison to the other GPU logic and fault simulation tools is not feasible due to our platform being significantly more advanced than the ones indicated in the literature; however, the achieved memory savings from our multiple mutant encoding scheme was able to increase the performance of our simulation kernel which was shown in the previous section. The first column lists the benchmark. Columns 2 through 6 lists the run-times of μ-*GSIM*

Table 5.2: Performance Analysis of *μ-GSIM* with Commercial Tool

Ckt.	Run-Time for *μ-GSIM*					Run-Time for Comm. Tool				
	Injected Mutants					Injected Mutants				
	1000	2000	3000	4000	5000	1000	2000	3000	4000	5000
b17	2.31	2.39	2.32	2.38	2.32	219.5	429.7	860.9	1149.5	1430.3
$b17_1$	1.62	1.57	1.53	1.57	1.63	214.6	524.3	887.4	1134.1	1347.1
b18	8.98	9.00	9.01	8.99	9.01	932.2	1851.6	2309.2	2862.3	3602.8
b20	0.76	0.76	0.76	0.76	0.76	266.3	579.7	864.7	1149.3	1418.8
b21	0.76	0.77	0.76	0.78	0.76	285.3	578.6	850.5	1135.5	1280.5
b22	2.22	2.22	2.23	2.22	2.22	280.1	505.4	623.0	948.8	1152.9

*Using MMGate Encoding, Reported run-times are in seconds

Table 5.3: Attained Speed-Ups for *μ-GSIM*

Speed-Ups (×) for *μ-GSIM* vs. Comm. Tool					
Ckt.	Injected Mutants				
	1000	2000	3000	4000	5000
b17	94.9	179.8	370.8	482.0	616.6
$b17_1$	132.3	334.2	579.5	724.1	828.7
b18	103.8	205.7	256.2	318.2	400.0
b20	349.6	762.5	1134.7	1506.2	1863.0
b21	374.0	752.2	1115.3	1462.5	1678.5
b22	126.2	227.9	279.7	428.0	518.3

for using the MMG encoding technique for the different number of injected mutants. As shown in Table 5.3, columns 2 through 6 lists the attained speed-ups against the commercial tool.

As we increase the number of mutations, the run-times for *μ-GSIM* remains virtually constant due to the fact there were a sufficient number of work-items launched on the GPU for concurrently simulating the entire set of mutant streams. The commercial tool experienced an increase in run-time for each successive injection of mutants by as much as 6× when increasing the number of mutants from 1000 and 5000. This shows that the simulation time required by the commercial tool is dependent on the number of mutated circuits that are simulated in a serial manner, whereas *μ-GSIM* simulates them in parallel.

We also observe that injecting 1000 mutations into the circuit yields a significant speed-up with our GPU tool. Additionally, the *μ-GSIM* run-times are visibly smaller compared to the commercial tool when simulating 5000 mutated circuits. The run-times ranged from twenty minutes to one hour whereas *μ-GSIM* required less than ten

seconds when looking into the largest circuit (`b18`). Uncovering the data independency of mutation testing, along with the proper memory consumption and mapping of mutation data, and leveraging the wealth of computational power of GPUs enables the evaluation of test sets in the least amount of time. This is favourable for large industrial circuits containing a plethora of test data.

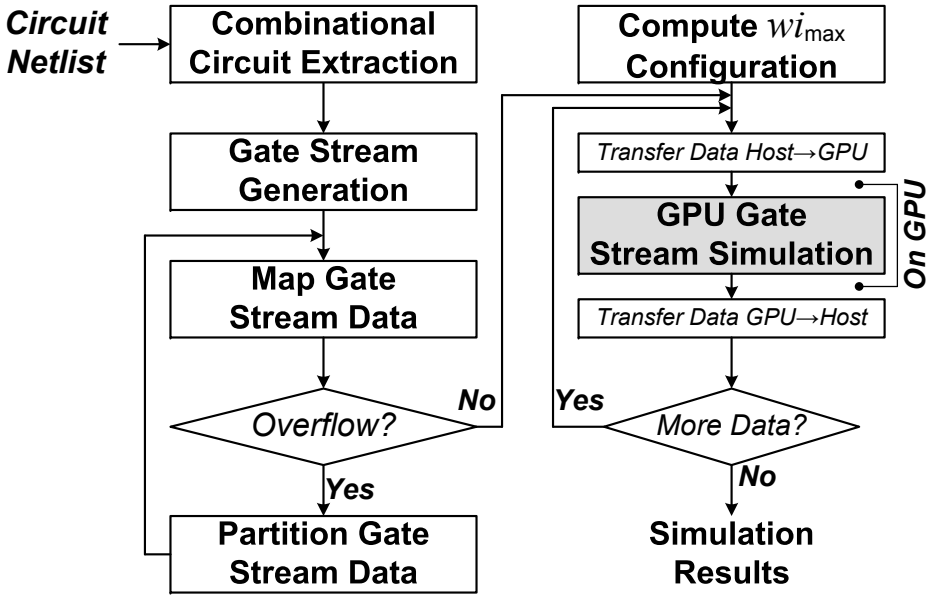


Figure 5.8: Overview of *GS-SIM*

5.3 *GS-SIM* Overview

This section presents *GS-SIM* which is based on the work of *μ-GSIM* that was presented in the last section. *GS-SIM* uses a similar encoding technique that encodes gate and fault data over the GPU's computer word.

Fault simulation requires a certain level of data dependency, which can pose a challenge in generating a data parallel representation of the circuit. We leverage the inherent bit parallelism of GPUs where we use the entire length of the computer word for representing the combined gate-fault and fault detectability data. Figure 5.8 shows the overview of our proposed *GS-SIM* simulator tool. The host program is responsible for generating the gate stream data, GPU device memory initialization and launching the simulation kernel. The GPU is responsible for fault simulating a set of gate streams concurrently, shown in the shaded box.

Many key factors must be exploited so that compute intensive applications (written either in OpenCL or CUDA) running on the GPU can achieve improved run-time performance [41]. One of them is to ensure all threads (work-items) access the GPU's device memory in a coalesced fashion, whereby multiple read and write accesses can be fulfilled by a single transaction. Threads within a warp (a group of 32 threads) must execute the same sequence of instructions so that branch divergence can be avoided. Lastly, threads should make effective use of the local memory available within the Graphics Processing Cluster (GPC) [39] for reducing the access latencies

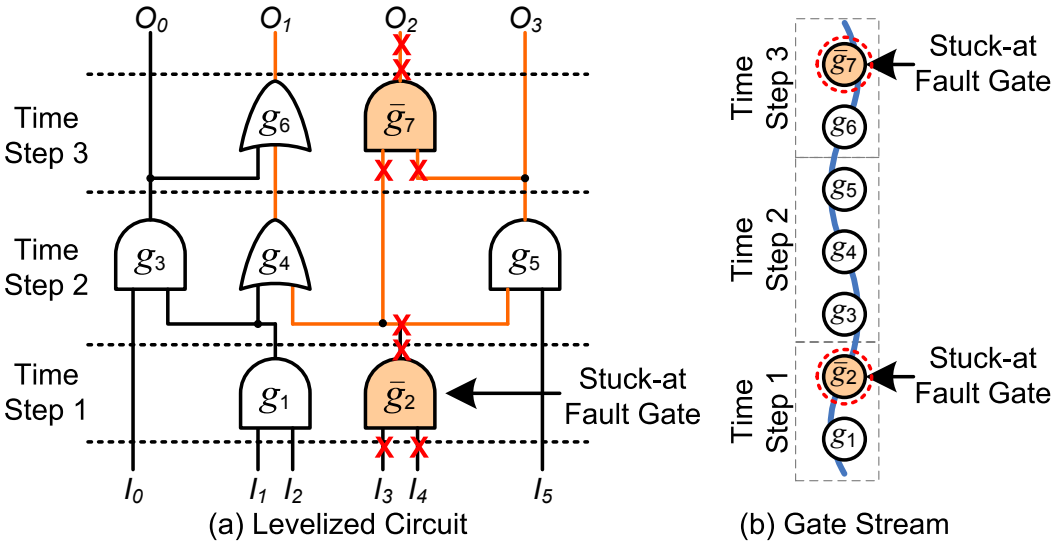


Figure 5.9: Combinational Circuit to Gate Stream Representation

to device memory. *GS-SIM* exploits all of these key factors in order to leverage the wealth of computer power of GPUs. We use the OpenCL programming language [40] so that our fault simulator is able to target other GPU architectures. The constructs and the execution model are similar to the CUDA C programming language [30] in which this was presented from Section 2.3.1.

5.3.1 Gate Stream Generation and MFG Encoding

GS-SIM uses the concept of gate streams, which is illustrated in Figure 5.9. This concept is similar to the one used in *μ-GSIM* for simulating multiple mutated circuits. Figure 5.9(a) depicts the circuit's gate-level netlist containing combinational logic elements. After applying the *As Soon As Possible* scheduling algorithm to the circuit, the set of gates from each time step is transformed into an array, as shown in Figure 5.9(b). We define this representation as a gate stream, gs_i, which is a one dimensional array of logic gates containing multiple stuck-at-fault gates, $\bar{g}_k$. Each stuck-at fault gate contains a set of collapsed stuck-at faults, F_k, where each fault $f_j \in F_k$ is either stuck-at-0 or stuck-at-1. The shaded nets starting from $\bar{g}_2$ depict the path for propagating the faults to the primary outputs O_1, O_2 and O_3.

The intention of the gate stream representation is to encode as many stuck-at fault gates in gs_i as possible so that a single work-item can simulate up to 31 faults. In the example, there are two fault gates, namely $\bar{g}_2$ and $\bar{g}_7$, as seen in Figure 5.9(a). They are represented by a 16-bit data word that is shown in Figure 5.10(a). The first

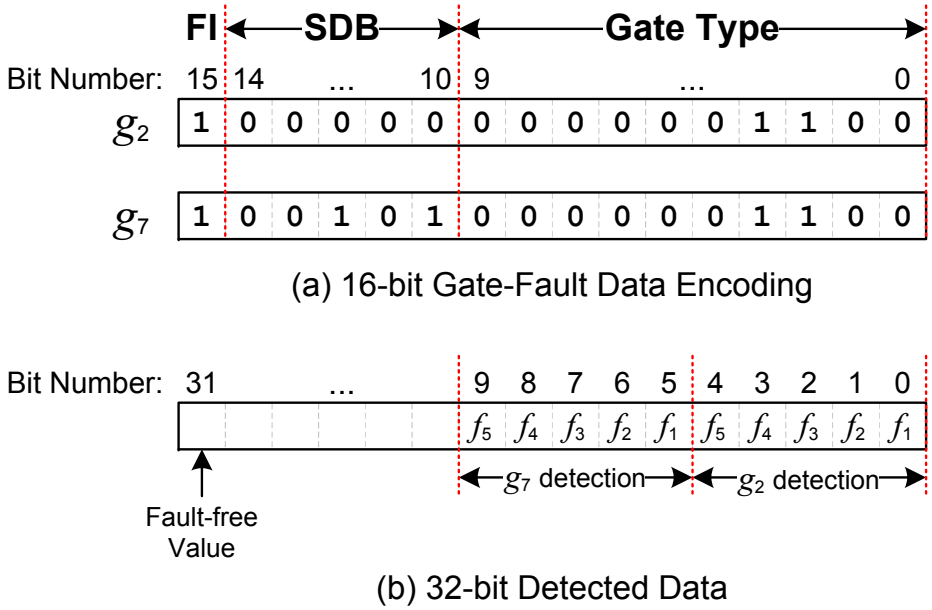


Figure 5.10: Gate-Fault and Detected Data Representation

10 bits of the word stores the gate type. A *Fault Injection* control flag, FI, is used for identifying a stuck-at fault gate $\bar{g}_k$ where the simulation kernel inserts the set of stuck-at faults, F_k. The *Shift Detection Bits* (shown as *SDB*) specify the number of left shifts for appropriately inserting the stuck-at faults over the 32-bit word during simulation. This value can range from 0 to 30 while leaving bit 31 as the fault-free value, which is used for computing the number of detected faults. As seen in the figure, $\bar{g}_2$ and $\bar{g}_7$ are the same gate type and both contain a set of faults causing their FI control flags set at logic-1. Without compromising the fault detection data during simulation, their shifted detection bits are set to 0 and 5 respectively. In the 32-bit detected word, $\bar{g}_2$ faults are assigned to bits 0 through 4 whereas $\bar{g}_7$ faults are assigned to bits 5 through 9, as shown in Figure 5.10(b). This encoding scheme can be expanded over a 32-bit data type if we use a 64-bit data variable for detecting faults or the number of gate types is greater than 1023.

5.3.2 Gate Stream Simulation of *GS-SIM*

The principle operation of a single work-item on the GPU is to perform fault simulation by leveraging several parallelism factors encoded within a single gate stream. Figure 5.11 outlines the different levels of parallelism that were achieved. First, we have exploited *gate parallelism* where every work-item is concurrently simulating the same set of gates and synchronizes at the end of each gate evaluation, which ensures

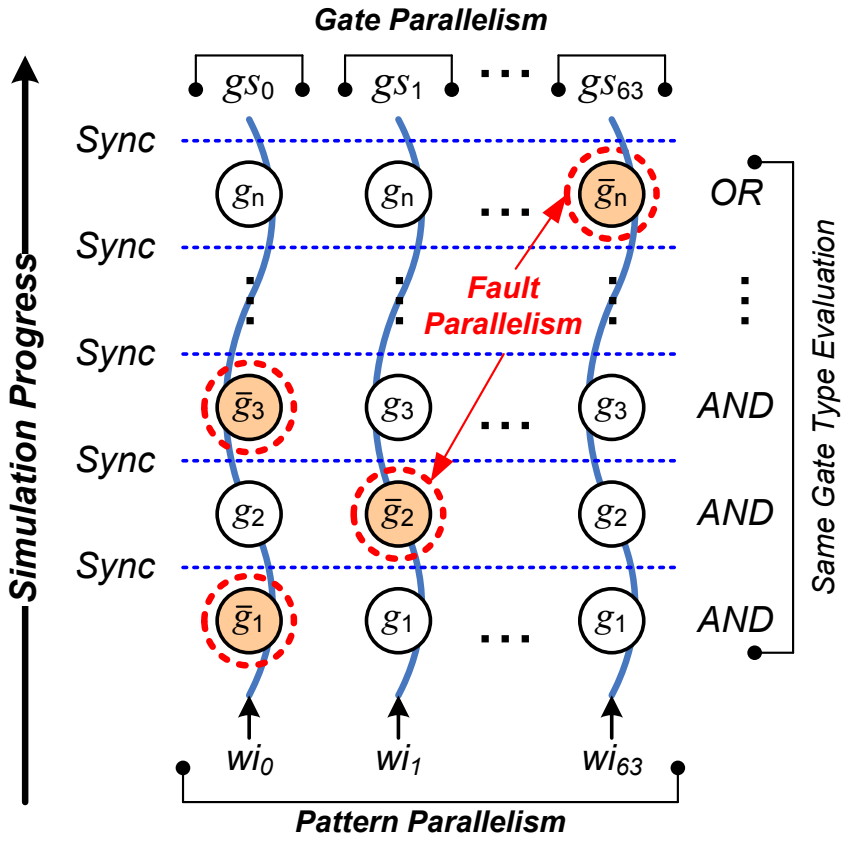


Figure 5.11: Exploited Parallelism Factors within 64 Gate Streams

that no divergent execution can occur. Second, *fault parallelism* was achieved because every work-item is fault simulating a unique set of stuck-at fault gates within a gate stream. Third, *pattern parallelism* was fulfilled by having every work-item simulate the same set of tests independently. Combining these levels of parallelism has enabled a group of 64 work-items (one work-item per gate stream) to fault simulate up to 1984 different stuck-at faults concurrently.

Algorithm 5.4 describes our proposed gate stream simulation kernel that is executed by every work-item on the GPU. The inputs to the kernel function are the generated gate stream data (*GS*), an array (*nets*) used for storing primary input, intermediate and primary output values during fault simulation and the detected fault data (*detected*). The input and output data are accessed through the GPU's device memory. The algorithm is a two phased approach.

At the beginning of the first phase, the `gate_stream_sim` kernel function retrieves the appropriate offset values which are accessible through the built-in OpenCL functions `get_global_size(0)` and `get_global_id(0)`. The index (0) returns the one dimensional ID value of the work-item. These offsets are used to compute the effective

Algorithm 5.4 Gate Stream Simulation Kernel

FUNCTION: gate_stream_sim
Input: Gate Stream Set (GS), Intermediate values ($nets$, Detected array ($detected$)

/* Retrieve gs offset values */
$num_{wi} \leftarrow$ **get_global_size**(0), $wi_{id} \leftarrow$ **get_global_id**(0)
$det\_local \leftarrow 0$ // Detected Fault Data
$gs_i \leftarrow GS[wi_{id}]$ // Retrieved Gate Stream

/* Retrieve gate-fault data g_k, then fault simulate */
for all $g_k \in gs_i$ **do**
  $gf\_data \leftarrow gs_i[g_k \times num\_wi + w_{id}]$
  gf_sim (gf_data, $nets$)

/* Compute Detected Data */
for all output ports p of gs_i **do**
  $t \leftarrow net_{31}$& 0xFFFFFFFF | $(1 - net_{31})$& 0x00000000
  $out \leftarrow p \times num_{wi} + wi_{id}$
  $det\_local \mid= t \oplus nets$[out]

address in GS so that every work-item accesses data from the GPU's device memory in a coalesced fashion. The variable, det_local, is used for storing the 32-bit detected data type in local memory of the GPC. This helps to reduce the number of read and write accesses to the device memory, because the detected data is continuously updated at the completion of a simulation run for a set of tests.

The variable gf_data stores the encoded 16-bit gate-fault data, and its state is determined by the function `gf_sim`. This function decodes the 16-bit variable for retrieving the gate type ($gate\_type$), the fault injection (FI) control flag, and the shifted detection bit (SDB). Then, the appropriate gate evaluation function is called based on the gate type. After all the gates are evaluated within the gate stream gs_i, the algorithm proceeds to compute the number of faults detected. This is done by retrieving the fault free value (bit 31) of $nets$ which gets expanded over a 32-bit word and stored into the variable t. Then, performing a logical `xor` operation such that the propagated fault can be detected at the circuit's outputs.

The second phase of the algorithm is to perform fault simulation of a logic gate using the appropriate gate evaluation function. An evaluation function is defined for every gate type that is supported in the gate library. For instance, the 10-bit coded $gate\_type$ `0000000100` represents a two input *and*-gate. The `gf_sim` function calls the

Algorithm 5.5 Evaluation Function for Two Input *and*-gate

FUNCTION: fsim_and2
Input: Input net addresses ($in1$, $in2$), Output address (out), $nets$, FI, SDB

SA0_mask ← LHS (SDB) of `0x01`
SA1_mask ← LHS ($SDB + 3$) of `0x0E`

/* If $FI = 1$ insert faults, otherwise no faults inserted */
SA0 ← 0xFFFFFFFF × (1-FI) | SA0_mask × FI
SA1 ← 0x00000000 × (1-FI) | SA1_mask × FI
in1_addr ← $in1 \times num_{wi} + wi_{id}$, v1 ← $nets$[in1_addr]
in2_addr ← $in2 \times num_{wi} + wi_{id}$, v2 ← $nets$[in2_addr]

/* Gate Evaluation */
out_addr ← $out \times num\_wi + wi_{id}$
$nets$[out_addr] ← ((v1 & v2) & (!SA0)) | SA1

evaluation function `fsim_and2`, which is shown in Algorithm 5.5. Evaluation functions for other gate types are similar.

Every gate evaluation function computes the fault-free value and the fault masks for injecting stuck-at faults. This eliminates the need for executing different kernel functions for computing the fault-free values and propagating the stuck-at faults separately. At the beginning of the evaluation function, the stuck-at-0 and stuck-at-1 fault masks are generated. The *and*2-gate type function generates the stuck-at fault masks by using the hard coded values `0x01` and `0x0E`, representing stuck-at-0 and stuck-at-1 faults respectively. Each fault mask performs a left-hand shift (LHS) of SDB bits, which appropriately inserts the set of faults over the 32-bit word without compromising the fault data from other stuck-at fault gates. Every stuck-at fault gate $\bar{g}_k \in gs_i$ will always have its faults inserted into the intermediate values as determined by the FI control flag. If the value of FI is 1, the faults are appropriately encoded over the 32-bit data word *SA0* and *SA1*, otherwise the fault-free masks are used.

Finally, the gate evaluation function will use the established input values for computing the output. Symbols $v1$ and $v2$ are the input values to the two input *and*-gate, followed by the logical *and* and *or* of the *SA0* and *SA1* values, which depict the insertion of the stuck-at-0 and stuck-at-1 faults respectively. Results are stored into the intermediate values array, $nets$, indexed in a manner such that the work-items write the results to the GPU's device memory in a coalesced fashion.

5.3.3 Maximum Work-item Configuration and Scalability

Determining the maximum number of work-items to deploy on the GPU depends on the available memory remaining (M_{rem}) in bytes after the gate stream data has been generated. The formulations that are presented in this section are based on the computations that were presented in Section 5.2.4 for the *μ-GSIM* tool. We define M_{tot} as the total memory in bytes that is required for simulating the entire set of gate streams:

$$M_{\text{tot}} = M_{\text{GS}} + M_{\text{nets}} \tag{5.6}$$

where M_{nets} is the memory for storing the intermediate values.

The required gate stream memory, M_{GS}, is defined by:

$$M_{\text{GS}} = (|gs| \times |GS|) \times \text{sizeof}(gf\_data) \tag{5.7}$$

where $|gs|$ represents the length of a gate stream in terms of the number of logic gates. Lastly, we multiply by the byte size of the 16-bit data word.

Based on the memory usage of M_{GS}, we initially compute the raw number of work-items that can potentially be instantiated as:

$$wi_{\text{raw}} = \frac{M_{\text{rem}}}{M_{\text{int}}}, \quad \text{with} \quad M_{\text{rem}} = M_{\text{gpu}} - M_{\text{GS}} \tag{5.8}$$

where M_{int} is the number of bytes used for storing the intermediate values in every gate stream; however, we require that the number of work-items be a multiple of 1024 so that each GPC can have at least 128 work-items. Thus, from wi_{raw} we derive the maximum number of work-items with:

$$wi_{\text{max}} = 1024 \times n, \quad \text{with} \quad n = \lfloor wi_{\text{raw}}/1024 \rfloor \tag{5.9}$$

We chose the floor operator when computing n so that the memory allocated for M_{nets} along with the gate stream data, M_{GS}, does not exceed the memory available provided by the GPU, M_{gpu}.

The computed maximum number of work-items affects how the gate stream data is mapped and simulated on the GPU. Different scenarios are shown in Figure 5.12. The host program aims to allocate as many work-items as possible so the entire set of gate streams can be simulated concurrently. This occurs when $wi_{\text{max}} \geq |GS|$ whereby each work-item is simulating a gate stream. Padding of the memory is needed in order to align the gate stream data to the number of work-items deployed. When the size of

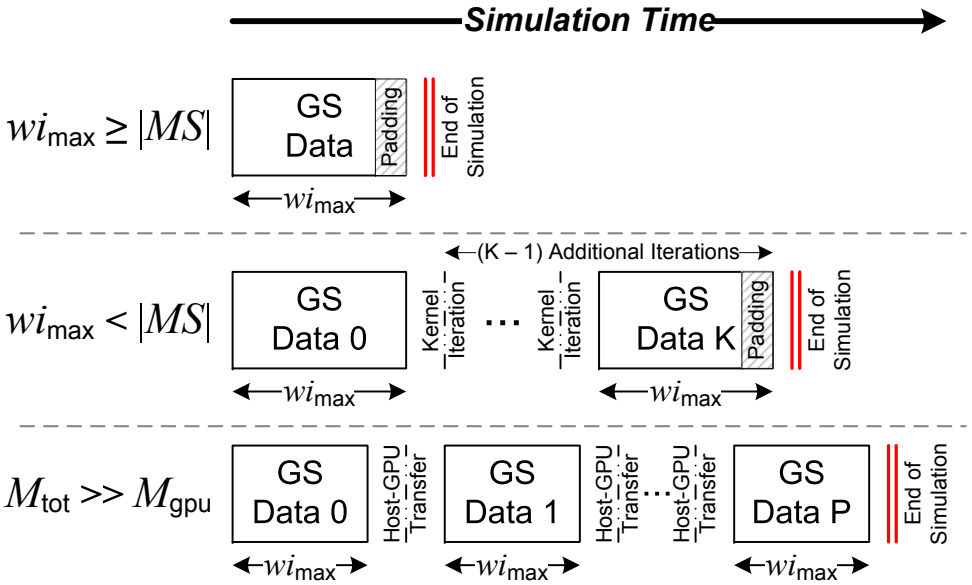


Figure 5.12: Gate Stream Data Mapping and Simulation Scenarios

M_{GS} is large, the host program may not allocate enough work-items to concurrently simulate a set of gate streams. This occurs when $wi_{\max} < |GS|$, thus requiring $(K-1)$ simulations, where K is defined as:

$$K = \left\lceil \frac{|GS|}{wi_{\max}} \right\rceil \tag{5.10}$$

Additional time is therefore required to complete the simulation. When the total memory greatly exceeds the memory capacity on the GPU ($M_{\text{tot}} \gg M_{\text{gpu}}$), the host program then partitions the gate stream data into P equal sets where each subset of gate streams is simulated separately. This requires more than one data transfer from the host to the GPU, which can decrease the overall fault simulation performance. Thus, it is imperative that the number of partitions be kept low.

The gate stream circuit representation combined with the multiple fault gate encoding are essential for creating an efficient mapping of the fault simulation data on a GPU. Our approach aims to allocate as many work-items on the GPU for simulating as many gate streams concurrently as possible.

5.3.4 Experimental Results of *GS-SIM*

We evaluated the performance of our *GS-SIM* tool against a CPU fault simulation tool on a set of large circuits from the ITC'99 benchmark suite [107]. Smaller benchmarks take negligible time and are not included due to space reasons. Our host (CPU)

Table 5.4: Total Memory Usage and Computed $wi_{\max}$

Circuit	**Gates**	**Encoding**	M_{GS} (MB)	M_{nets} (MB)	M_{tot} (MB)	$wi_{\max}$
b17	30777	SFG	1806.7	251.5	2058.2	1024
		MFG	230.7	628.8	859.5	**5120**
b17_1	38166	SFG	2778.3	154.4	2932.7	1024
		MFG	372.2	463.3	835.5	**3072**
b18	111241	SFG	3906.6	447.5	4354.1	1024
		MFG	563.7	447.5	1011.2	1024
b19	224624	SFG	5786.5	898.5	6684.9	1024
		MFG	1041.6	898.5	1940.0	1024
b20	19682	SFG	738.9	157.6	896.5	2048
		MFG	269.1	630.4	899.5	**8192**
b21	20027	SFG	765.0	160.3	925.3	2048
		MFG	391.2	561.0	952.2	**7168**
b22	29162	SFG	1622.1	116.8	1738.8	1024
		MFG	398.7	467.1	865.8	**4096**

platform is a six-core AMD processor running at 3.2GHz with 16GB of memory. The GPU device is a GTX 560 Ti graphics card containing 8 GPCs, each consisting of 48 processing cores with a core clock speed of 1.7GHz and is equipped with 1GB of device memory. We used the NVIDIA CUDA SDK 4.2.1, using version 1.1 of the OpenCL language.

Table 5.4 shows the memory usage and the computed $wi_{\max}$ values that can be launched onto the GPU. For each circuit benchmark, we applied the Single Fault Gate (SFG) encoding (one stuck-at fault gate per gate stream) and Multi-Fault Gate (MFG) encoding scheme and set the value of M_{GPU} to 960MB. As seen in the table, the MFG encoding of the circuits yields a decrease in the M_{GS} memory requirement by nearly 80%, with an associated increase in the number of work items $wi_{\max}$, implying that more gate streams can be concurrently simulated. Some circuits do not show an increase in the number of work-items due to their large size. The gate stream data for those circuits are partitioned into equal sets where additional host to GPU transfers are required.

Table 5.5 shows the average run-time comparisons when employing the SFG and MFG encoding schemes with our *GS-SIM* tool. We also compare our work to an open source fault simulator tool, *FSIM* [108], which ran on the CPU. A direct comparison to the other GPU fault simulation tools is not feasible due to our GPU hardware being more advanced than the ones indicated in the literature; however, the memory savings

Table 5.5: Performance Analysis of *GS-SIM*

Circuit	**Run-times** (s)			**Speed-up** (×) **vs FSIM** [108]		**Speed-up** (×)
	MFG	SFG	*FSIM*	MFG	SFG	MFG vs SFG
b17	0.40	1.50	23.83	60.2	15.9	3.8
b17_1	0.49	1.85	27.09	55.3	14.6	3.8
b18	4.42	11.05	138.97	31.4	12.6	2.5
b19	5.74	14.92	319.00	55.6	21.4	2.6
b20	0.16	0.47	4.98	31.2	10.5	3.0
b21	0.13	0.48	5.56	41.5	11.5	3.6
b22	0.36	1.42	9.49	26.5	6.7	4.0
			Average:	43.1	13.3	3.3

achieved can benefit other approaches as our encoding technique is independent of the GPU architecture. We fault simulated each circuit using 32K (32,768) random tests and performed ten simulation runs to ensure that the measured run-times are consistent. The number of work-items launched are based on the computed $wi_{\max}$ values, shown in Table 5.4.

The average run-times from our tool when employing the MFG encoding are visibly reduced compared to the SFG encoding as shown in columns two and three. The decrease in memory usage has led to an average speed-up of 3.3×, with a maximum of 4.0× compared to the SFG encoding, as shown in the rightmost column. This is due to the number of work-items that were launched onto the GPU is sufficient for simulating the entire set of gate streams concurrently. The large circuit benchmarks also show a decrease in run-time. This is caused by the reduced memory usage for storing gate stream data, which in turn leads to a decrease in the number of data transfers between the host and the GPU.

Both encoding techniques achieve significant performance gains when compared to *FSIM*, with average and maximum speed-ups of 43.1× and 60.2× respectively, for the MFG encoding (column 5). This shows that memory efficiency and the exploitation of the many parallelism factors of GPUs are key for improving application performance.

5.4 Summary

Graphics processing units are gaining in popularity for parallelizing many forms of computations, including accelerating simulation algorithms for the validation of digital circuit designs. One of the challenges is to determine an efficient mapping of the

circuit data so that every work-item can perform its simulation independently. This chapter proposed two GPU-based tools, namely *μ-GSIM* and *GS-SIM* for accelerating mutation-based and fault simulation algorithms on the GPU. Both of the tools have leveraged the inherent bit parallelism of GPUs; we showed how to exploit gate, mutant, fault and pattern parallelism factors within a mutant and gate streams of a circuit. The novel concepts of the Multiple Mutant Gate (MMG) and Multiple Fault Gate (MFG) encoding schemes has helped in reducing memory consumption of the required circuit data, which had a positive impact on the simulation performance in our proposed simulation tools. The proposed memory encoding techniques we put forth allowed for larger circuits to be handled, which is essential for validating and assessing test quality of modern circuit designs.

Chapter 6

Using GPUs for Accelerating Mutation Testing of Assertions

This chapter presents μDV-GSIM, an accelerated mutation testing GPU framework for assessing test quality from assertions. The proposed framework is based on *μ-GSIM*, which was presented in Section 5.2. Table 6.1 highlights the differences between *μ-GSIM* and the proposed μDV-GSIM that is presented in this chapter. The proposed framework also uses the efficient memory encoding techniques, that were presented in Chapter 5, for generating an effective data-parallel representations of multiple mutated designs onto the GPU's device memory. μDV-GSIM utilizes assertion-based tests that are encoded over the GPU's computer word. This enables one work-item using 32 different assertion-based tests concurrently, which can be simulated with multiple mutated designs. Secondly, mutations are injected at the design (RTL) level, and are synthesized into their equivalent gate-level representation. Finally, by encoding the assertion-based tests, Assertion-Test parallelism was exploited with μDV-GSIM. The goal of this chapter is to present a novel method for accelerating mutation testing of assertion-based tests, in the effort for reducing the computational time. These concepts are presented in the subsequent sections.

Table 6.1: Comparison between *μ-GSIM* and μDV-GSIM

Category	*μ-GSIM*	μDV-GSIM
Applied Tests	Random	Assertion-based
Data Encoding Type	Multiple Mutants	Multiple Assertions
Injected Mutant	Gate-level	RTL-level
Circuit Representation	Mutant Stream	Circuit Stream
Parallelism	Gate, Mutant	Gate, Assertion-Test

6.1 Motivation

Mutation testing is a technique that is based on software testing, which is used for gauging the quality of tests. It can also be used for gauging the quality of assertions, by having the assertion-based tests simulate multiple mutated designs, each containing an injected functional fault. At the end of simulating the mutated designs, the amount of mutations that were undetected (survived) can be used in developing additional assertions. This leads to a strengthened set of assertions that is capable of uncovering other faults.

Mutation testing is computationally expensive, because of the amount of simulation runs that are needed for simulating multiple mutated designs. Additionally, as designs increase in size and complexity along with their test data, the computational cost would outweigh the benefits of using this technique for assessing test quality. As shown from Section 2.2, analyzing the quality of assertion-based tests using mutation testing can benefit from the massive parallelism provided by GPUs, as also explained from the GPU architecture in Section 2.3. Since each functional fault (mutant) is independent of each other, then multiple mutated designs can be simulated concurrently, thereby accelerating the mutant simulation process. Furthermore, the set of assertion-based tests can also be used simulatenaously by all the mutated designs.

In this chapter, μDV-GSIM is presented. This is a GPU-based tool that uses mutation testing for assessing test quality of assertions. The assertion-based tests generated from the developed framework in Chapter 3 will be used. μDV-GSIM attempts to exploit many levels of parallelism so that the tool is able to generate an efficient mapping of the multiple mutated design data and assertion-based tests in order to gauge the completeness of a given assertion set. Additionally, the proposed optimized mapping technique has contributed a speed-up by a factor of up to 2.5× in additional performance improvements with a speed-up of 10.6× on the largest benchmark, compared to when having each thread simulate a set of tests from one assertion.

The contributions brought forth into this chapter are summarized as follows: [1]

- A GPU-based simulator was developed for assessing the quality of tests derived from assertions;
- An encoding scheme was developed which is used for storing multiple mutated design and assertion test data, which leads to improved simulation performance

[1] The contents of this chapter is based on the article entitled *Using GPUs for Assessing Assertion Quality*

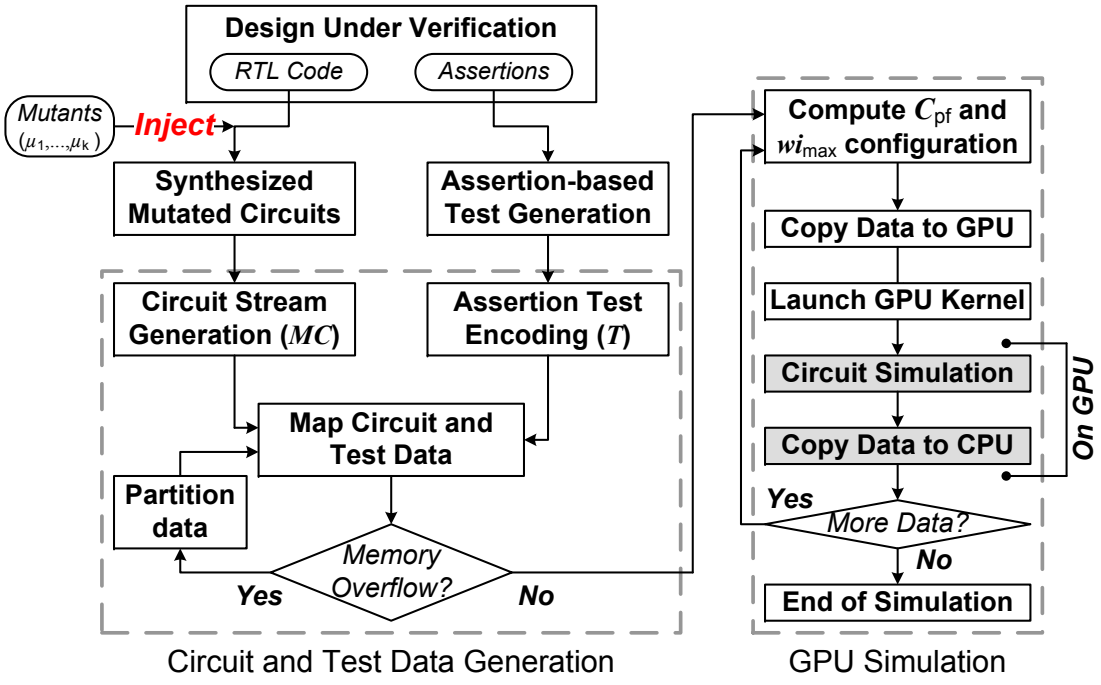


Figure 6.1: μDV-GSIM Framework

on the GPU

6.2 μDV-GSIM Overview

Figure 6.1 shows the overview of the μDV-GSIM framework. μDV-GSIM exploits the inherent bit parallelism of the GPU where the entire length of the GPU's computer word is used for storing multiple assertion-based test data along with the detected assertion data. Our framework takes in a Design Under Verification (DUV) where the design is given in Register Transfer Level (RTL) code and a set of assertions that verifies the correctness of the design. A set of mutants, F, is also provided where each mutant operator μ depicts a fault at the RTL level. We focus on *synthesizable* mutations and using the five key mutants that were defined in [26]. Each mutant, μ_k, gets inserted into the RTL code and is synthesized into a gate-level mutated circuit, C_k , by a synthesis tool. The set variable MC stores the synthesized gate-level circuits:

$$MC = \{C_0, C_1, C_2, ..., C_k\}$$

where C_0 represents the reference (mutant-free) circuit. The assertions that are provided undergo an *Assertion-based Test Generation*.The defined sequence of events

from each assertion, ϕ_i , gets transformed into a test sequence by our TG tool and are stored into the set variable, $T_{\phi,i}$. Each test, $t_{j,i} \in T_{\phi,i}$, contains the appropriately assigned Boolean values for each defined primary input of the design. The entire set of the assertion-based tests gets stored into the set variable:

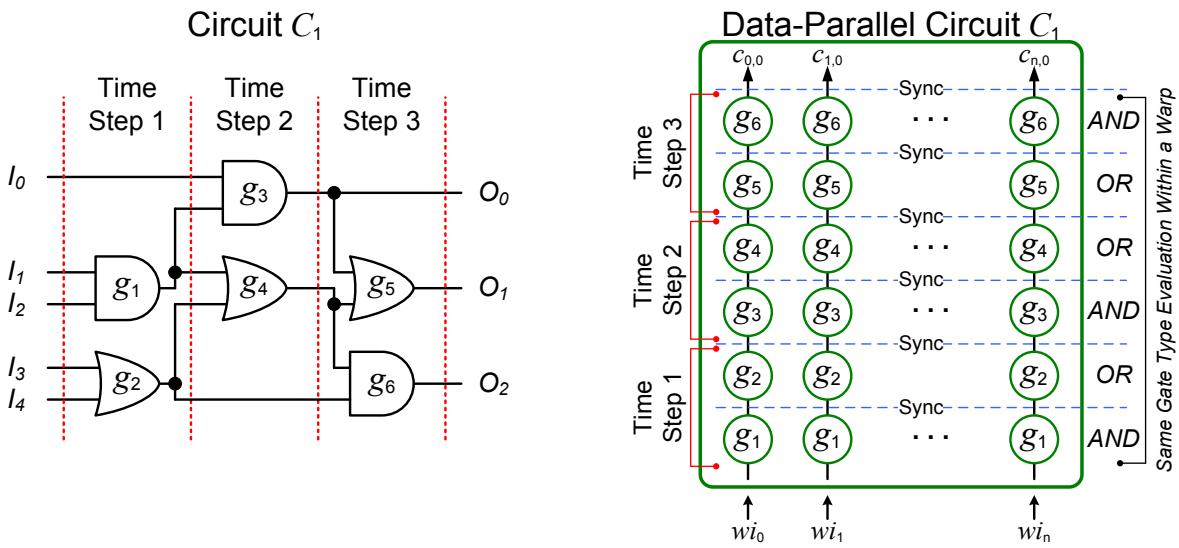
$$T_\phi = \{T_{\phi,0}, T_{\phi,1}, ..., T_{\phi,n}\}$$

for all of the n assertions from A.

After generating the set of synthesized mutated circuits MC and the assertion-based tests T_ϕ, the next step is to generate an efficient data parallel representation of the mutated circuit and test data. This is performed during the *Circuit Stream Generation* and the Assertion Test Encoding stages of the μDV-GSIM tool. During these stages, the circuit and test data are effectively duplicated so we can have multiple work-items on the GPU simulating many mutated circuits independently. In the event of a memory overflow while mapping the circuit and test data, the host program will partition the set of mutated circuits into equal portions and will then perform a separate kernel execution call. The second phase is the *GPU Simulation*, where the host program determines the maximum number of work-items ($wi_{\max}$), which leads to computing the *Circuit Parallelism Factor* (C_{pf}). These computed values are used for allocating the work-items on the GPU. In the next sections, we thoroughly describe these phases.

6.3 Circuit Stream Generation

μDV-GSIM uses the concept of *circuit streams* for simulating multiple synthesized mutated circuits on the GPU. A *circuit stream*, defined as $c_{i,k} \mid i = 0, 1, 2, ..., n_s$, where n_s represents the number of circuit streams inside each C_k. Each circuit stream $c_{i,k}$ depicts a one dimensional array of logic gates ($g \in c_{i,k}$) that are arranged in a particular order so that their input and output dependencies are satisfied. Figure 6.2(a) shows an example of a levelized mutated circuit netlist, C_1. After applying the *As Soon As Possible* scheduling algorithm to the circuit, the set of gates from each time step gets transformed into a data-parallel arrangement, as shown in Figure 6.2(b). Each labelled node depicts a specific gate inside the circuit. The *Multiple Circuit Stream* representation for the synthesized mutated circuit C_1 enables every work-item to evaluate the *same gate types* in a lockstep manner. This prevents any divergent execution to occur within a warp of work-items, which is one of the key factors when



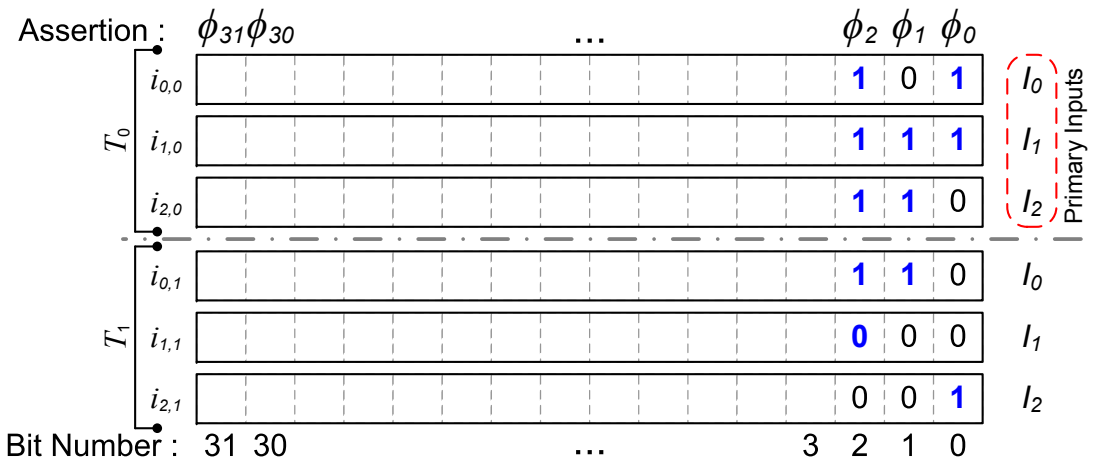
(a) Levelized Circuit (b) Multiple Circuit Stream Representation

Figure 6.2: Circuit Stream Representation

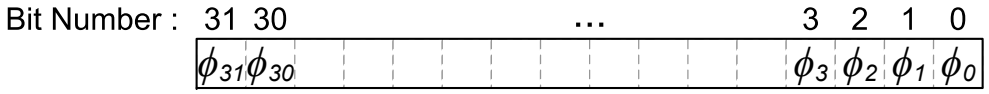
programming GPUs. Work-items are assigned to a particular circuit stream that is based on its indexing value. Additionally, each circuit stream is assigned to a particular set of tests so that every work-item is simulating a circuit using different tests independently. The multiple circuit stream representation is also applied to every synthesized mutated circuits $C_k \in MC$.

Circuit streams employed by μDV-GSIM differ from the previous mutant and gate streams that were used in *μ-GSIM* and *GS-SIM* respectively. In both of those approaches use a single gate-level netlist for generating an efficient data parallel representation while every mutant and manufacturing fault was injected at the gate-level. Since μDV-GSIM simulates synthesized mutated circuits where the mutant μ_k is injected at a higher level of abstraction (RTL level), then every circuit C_k can potentially have different circuit structures with each other. This is caused by the gate-level optimizations that were performed by the synthesis tool. Circuit streams belonging to different mutated circuits $(c_{i,a}, c_{i,b} \mid a \neq b)$ will have different gate types at every position in the array $(g_{i,a} \neq g_{i,b})$. To ensure that every warp of work-items evaluate the *same circuit structure* in terms of their gate types, the assigned number of work-items per synthesized circuit should be a multiple of 32. This implies that the number of circuit streams n_s in every synthesized circuit C_k must be a multiple of 32 ($n_s \ mod \ 32 = 0$).

The value of n_s is dependent on the number of assertion-based tests in T_ϕ. For instance, if each work-item simulates a single subset of tests $T_{\phi,i}$ from assertion ϕ_i,



(a) Encoded Test Packets for wi_0



(b) Detected Assertion Data for $c_{0,10}$

Figure 6.3: Multiple Assertions Encoding

then the minimum number of circuit streams per circuit is $|A|$; however, since each subset of tests are independent of each other while the synthesized mutated circuit is decomposed into their individual bit members, then the GPU's computer word is leveraged for encoding multiple subset of tests that can be used by a single work-item. This can significantly reduce the required number of streams to its minimum value of 32. In the next section, the proposed encoding technique, namely Multiple Assertions Encoding, that describes how the tests are stored over the length of the GPU's computer word.

6.4 Multiple Assertion Encoding (MAE)

Every work-item wi_{i} on the GPU will evaluate a circuit stream $c_{i,k}$ belonging to a respective circuit, C_k. Additionally, each circuit C_k will have work-items applying their assigned test sequences that were generated from assertions, which implies one work-item is simulating a set of tests that was derived from an assertion ϕ_i. For example, work-item wi_0 can be simulating circuit stream $c_{0,4}$ that belongs to circuit C_4, while applying test sequences that were derived from assertion ϕ_0. Since the synthesis tool decomposes every logic element into their individual bit members, then

the GPU's 32-bit computer word is used for assigning different Boolean logic *test values* at specific bit positions for every primary input. This enables one work-item to simulate tests from 32 different assertions. Thus, creating the potential of having up to 1024 different assertion-based tests with a circuit containing 32 circuit streams (32 assertions per work-item $\times$ 32 work-items per circuit). This concept is defined as the *Multiple Assertion Encoding* (MAE) which is a technique for reducing the number of streams n_s to its minimum value while also being able to simulate as many assertions concurrently as possible.

Figure 6.3 shows an example of the MAE concept that is used in μDV-GSIM for encoding multiple assertion-based tests. During the *Assertion Tests Encoding* stage, the generated set of tests has their individual bit members encoded over a 32-bit word. For instance, consider the subset of assertions that are defined below where the signals references the primary inputs of the circuit :

ϕ_0: `I0 && I1 |=> I2`

ϕ_1: `I1 && I2 |=> I0`

ϕ_2: `I0 && I1 && I2 |=> I0 && !I1`

The assertion-based test generator will take in those assertions and generate the necessary test sequences by assigning the appropriate Boolean logic values. It is assumed that a value of logic-0 is assigned when a signal condition is not explicitly defined in the assertion. Then, the MAE encoding scheme will transform those tests and appropriately assign logic values at specific bit positions over the entire 32-bit word, as depicted in Figure 6.3(a). Each input test sequence T_s has a corresponding set of *test packets* $i_{p,s}$ where packet i_p belongs to test, T_s. The number of input packets per test is equal to the number of primary input ports of the circuit.

Every assertion is assigned to a bit position over the 32-bit word. Assigning their corresponding Boolean logic values are determined by the specified signalling conditions of the assertion. For example, assertion ϕ_2 is assigned at bit position 2 of the 32-bit word. The first input test sequence (T_0) requires signals `I0`, `I1` and `I2` be at logic-1. Consequently, a logic-1 is assigned at bit position 2 for test packets $i_{0,0}$ through $i_{2,0}$ belonging to input test, T_0. This is indicated by the blue text in the figure.

Due to every assertion having the non-overlapping operator (`|=>`), a next test is created, which is T_1. In the consequent portion of assertion ϕ_2, the input signals `I0` and `I1` is required to be at logic-1 and logic-0 respectively. This is indicated in the input test packet $i_{0,1}$ and $i_{2,1}$ at bit position 2.

Figure 6.3(b) shows the 32-bit word that stores the detected assertion data for

work-item, wi_0. Each bit position corresponds to the assigned assertion ϕ_i. When the test set $T_{\phi,b}$ causes the mutated circuit C_k to produce an output that differs from the reference design, then assertion ϕ_b is said to have *detected* the mutant μ_k. Then, the corresponding assigned detection bit b for the assertion in detected the mutant, ϕ_b, is set to logic-1. This implies that the assertion is able to detect the injected mutant. Otherwise, leaving the assigned detected bit at logic-0.

It is noted that the above MAE example shows the test encoding for work-item wi_0. This concept also applies for other work-items where work-item wi_1 is using tests derived from assertions ϕ_{32} through ϕ_{63}, and so forth. For $n_s = 32$, then each synthesized circuit C_k can simulate up to 1024 assertions concurrently. When the assertion set is greater than 1024, then the host program will increase the number of circuit streams by a factor of 32, thereby each circuit will contain 64 circuit streams and can simulate using 2048 assertions.

The mutated circuit stream approach along with the multiple assertion encoding scheme presented in this section will now be used for simulating multiple mutated circuits on the GPU. The next section describes the GPU simulation phase of the μDV-GSIM.

6.5 Circuit Stream Simulation

In this section, the description of the proposed GPU circuit simulator is presented for assessing test quality from assertions on multiple synthesized mutated circuits.

6.5.1 Simulation Kernel

Figure 6.4 shows one of the memory organizations for mapping 32 circuits inside the GPU's device memory. The primary function for each work-item on the GPU is to perform logic simulation on the mutant and mutant-free (C_0) circuits using their assigned assertion-based tests. As seen in the figure, several parallelism factors have been achieved from the transformed data-parallel representation of the mutated circuit and test data. First, *gate-level parallelism* was exploited, where each mutated circuit containing 32 circuit streams ($n_s = 32$) and are being simulated by 32 work-items. Each group of work-items in every circuit will evaluate the same gate-type and synchronizes at the end of each evaluation. This ensures that every warp of work-items on the GPU will have the same instruction flow, thereby avoiding divergent execution. Second, *mutant parallelism* was exploited because in each set of 32 circuit

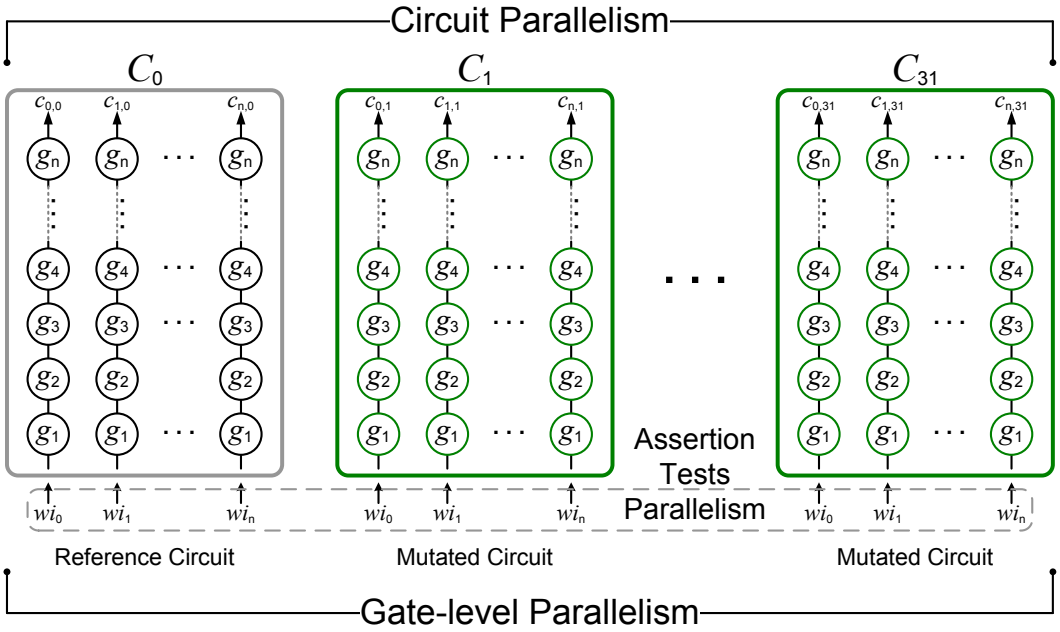


Figure 6.4: Parallelism Factors

streams (excluding the reference circuit C_0) contains a distinctive synthesized mutant and are simulated independently. Third, *assertion tests parallelism* was fulfilled where each work-item is using its assigned tests that are applied to the circuit's primary inputs.

Algorithm 6.1 shows the proposed simulation kernel that operates on a circuit stream belonging to a synthesized circuit, C_k. This algorithm is similar to the ones that are shown in the *μ-GSIM* and the *GS-SIM* GPU simulators. The inputs to the kernel function are the generated circuit stream data (CS), an intermediate value array ($nets$) for storing the input and output values for every logic gate and sequential elements (flip-flops), the detected assertion data ($detected$) and the encoded test data stored (T_s). These are accessed through the GPU's device memory and are accessed in a coalesced manner. The variable n_s is a constant variable indicating the number of circuit streams in every circuit, C_k.

At the beginning of the algorithm, the kernel retrieves the appropriate offset values which are accessible through the built-in OpenCL kernel functions, `get_global_size(0)`, `get_global_id(0)` and `get_local_id(0)`. The index parameter `(0)` returns the one dimensional ID value of the work-item. The global work size and ID values are used for accessing the circuit stream data and the intermediate values in a coalesced fashion, whereas the local ID is used for retrieving the encoded test data and computing the detected assertion data. Using the appropriate offset values, the kernel retrieves

Algorithm 6.1 Circuit Stream Simulation Kernel

FUNCTION: circuit_stream_sim
Input: Circuit Stream Set (CS), Intermediate values ($nets$), Detected array ($detected$), Tests (T_s), Streams Per Circuit (n_s)

/* Initialize Variables*/
$num_{wi} \leftarrow$ **get_global_size**(0) // Total Num Work-items
$wi_{gid} \leftarrow$ **get_global_id**(0) // Global Work-item ID
$wi_{lid} \leftarrow$ **get_local_id**(0) // Local Work-item ID
$det\_local \leftarrow 0x00000000$ // Detected data
$c_{i,j} \leftarrow CS[wi_{gid}]$ // Circuit Stream

for all $i_{p,s} \in T_s$ **do**

  /* Initialize Input Ports with Test Packets */
  for all input ports $p \in c_{i,k}$ **do**
    $nets[\text{p} \times num_{wi} + wi_{id}] = i_{p,s}$

  /* Update Sequential Elements */
  for all Present state nets $\in c_{i,k}$ **do**
    Transfer Next State nets $\in c_{i,k}$ to Present State nets

  for all logic gates $g_j \in c_{i,k}$ **do**
    /* Pass gate type to gate_sim kernel */
    $gate\_type \leftarrow g_j$
    gate_sim ($gate\_type$)

  /* Compute Detected Data */
  for all output ports p of $c_{i,k}$ **do**
    $ref \leftarrow nets[p \times num_{wi} + wi_{lid}$ // Reference Circuit Output
    $det\_local \mid= (nets[p \times num_{wi} + wi_{id}] \oplus ref)$

/* Store Detected Data to Device Memory */
$detected[wi_{id}] \leftarrow det\_local$

the circuit stream data and stores it into the variable, $c_{i,k}$ where i is the work-item local ID and k is the appropriate offset for accessing circuit, C_k. To further reduce the number of accesses to global memory, the 32-bit detected assertion data is stored in the GPC's local memory because the variable *det_local* is continuously updated at the completion of simulating every input test sequence, T_s. Additionally, local memory provides lower access latencies (one clock cycle) compared to global memory (400-800 clock cycles).

Algorithm 6.2 Gate Evaluation Function for Three Input *and*-gate

FUNCTION: and3_gate_sim
Input: Input net addresses ($in0$, $in1$, $in2$), Output net address (out)

/* Retrieve Input Values */
$in0\_v \leftarrow nets[num_{wi} \times in0 + wi_{id}]$
$in1\_v \leftarrow nets[num_{wi} \times in1 + wi_{id}]$
$in2\_v \leftarrow nets[num_{wi} \times in2 + wi_{id}]$

/* Compute and Write Output Result to *nets* */
$nets[num_{wi} \times out + wi_{id}] \leftarrow \mathbf{and}(in0, in1, in2)$

Each circuit stream in every mutated circuit gets simulated by the assigned assertion-based tests. Every work-item will use their local IDs for retrieving the input test packet data so that work-item wi_0 will use test packets $i_{p,0}$ from T_0 , then work-item wi_1 will load test packets from T_1 and so forth. Test packets are loaded into the primary input locations in *nets*. Then, the next state inputs are transferred to the present state outputs, which depicts the behaviour of the sequential elements inside the circuit.

For every gate g_i inside the circuit stream $c_{i,k}$, its 16-bit data word is retrieved then stored into the variable, *gate_type*. The gate type data is used by the function *gate_sim*, which invokes the appropriate gate evaluation function. During this state, the kernel will then perform logic simulation on all the gates inside the circuit stream.

At the completion of simulating the circuit stream, the detected assertion data is computed. This is done by using the work-items local ID for retrieving the primary outputs values from the reference circuit, C_0, and stored into the variable, ref. Using the output values from the mutated circuit along with the reference output values, then a logical *xor* operation is performed and its result is stored into local memory through the variable *det_local*. Then, the next test (T_{s+1}) is used for simulation and repeats until all tests have been simulated.

The function `gate_sim` was called from Algorithm 6.1, which is used for invoking the appropriate gate evaluation function. For example, the value of *gate_type* is `0x000E` which is the 16-bit gate code of a three input *and*-gate. Then, `gate_sim` invokes the `and3_gate_sim` gate evaluation function which is listed in Algorithm 6.2. Evaluation functions for other gate types are similar to the one presented. Every gate evaluation function will compute the value of the logic gate's output based on the assertion-based tests that are supplied at the primary inputs or the intermediate values. Since each circuit stream $c_{i,k}$ belonging to a circuit C_k has its set of logic gates

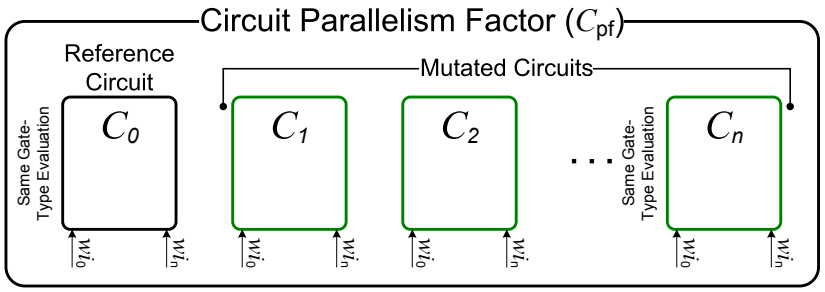


Figure 6.5: Circuit Parallelism Factor

arranged with the same gate-types in every position of the array, then every work-item within a warp will execute the same gate evaluation function. Furthermore, with every work-item accessing the same rows in the *nets* variable in global memory using continuous address offset based on their global work ID, then the work-items within a warp will read from and write to global memory in a coalesced fashion.

6.5.2 Circuit Parallelism Factor and Memory Scalability

Determining the number of synthesized mutated circuits that can be concurrently simulated on the GPU depends on the maximum number of work-items that can be deployed. The *Circuit Parallelism Factor*, defined as C_{pf}, depicts the number of circuits that can be simulated within a single iteration of the simulation kernel, as depicted in Figure 6.5. This value determines the number of kernel iterations that are needed in order to simulate the entire set of mutated circuits. It is favourable if this value can exceed, $|MC|$, so that one kernel iteration is needed to simulate the entire set of mutated circuits. The CPF value is computed by:

$$C_{pf} = \frac{wi_{max}}{n_s} \tag{6.1}$$

where $wi_{\max}$ is the maximum number of work-items that can be allocated on the GPU and n_s is the number of circuit streams for every circuit, C_k. To achieve a high C_{pf} value, it is imperative that the number of circuit streams in each circuit C_k be at its minimum while the host program aims to allocate as many work-items on the GPU as possible. As described from Section 6.3, the number of circuit streams in every circuit must be a multiple of 32. Thus, the minimal value of n_s should be kept at 32 where possible.

Determining the maximum number of work-items to deploy on the GPU depends on the available memory remaining after mapping the circuit stream data, CS. Com-

puting $wi_{\max}$ is dependent on the memory usage by the circuit stream data M_{CS} and the amount of bytes for storing the intermediate values M_{int} per work-item. The total memory, M_{tot}, is the memory required for simulating the entire set of circuit streams and it is computed by:

$$M_{tot} = M_{CS} + M_{nets} \tag{6.2}$$

The memory required for storing the circuit stream data of every circuit, M_{CS} is defined by:

$$M_{CS} = |c_{i,k}| \times n_s \times |MC| \times sizeof(gate\_type) \tag{6.3}$$

where $|c_{i,k}|$ is the length of the circuit stream in terms of the number of logic gates, n_s is the number of circuit streams per circuit and $|MC|$ is the number of synthesized circuits required for simulation.

Initially, the raw number of work-items is computed as the following:

$$wi_{\text{raw}} = \frac{M_{\text{rem}}}{M_{\text{int}}}, \quad \text{with} \quad M_{\text{rem}} = M_{\text{gpu}} - M_{\text{CS}} \tag{6.4}$$

where M_{int} is the number of bytes used for storing the intermediate values in every gate stream; however, it is required that the number of work-items be a factor of 1024 so that there are at least 128 work-items in each Graphics Processing Cluster (1024 work-items / 8 Graphics Processing Clusters) simulating 4 mutated circuits (if $n_s = 32$, then 1024 / 32 work-items per circuit = 32 mutated circuits). Thus, from wi_{raw}, the maximum number of work-items is computed as:

$$wi_{\max} = 1024 \times n, \quad \text{with} \quad n = \lfloor wi_{\text{raw}}/1024 \rfloor \tag{6.5}$$

The floor operator is chosen when computing n so that the memory allocated for M_{nets} along with the circuit stream data, M_{MC}, does not exceed the memory available provided by the GPU (M_{gpu}).

Computing the maximum number of work-items is limited to the memory consumed by the circuit stream data and the memory required for storing the intermediate values. This directly affects the circuit parallelism factor which determines the number of circuits that can be simulated concurrently with one kernel iteration, thereby affecting the overall simulation performance. Figure 6.6 shows the different data mapping scenarios and the flow of simulation. The host program aims to allocate as many work-items as possible in order to simulate the entire set of circuits MC with-

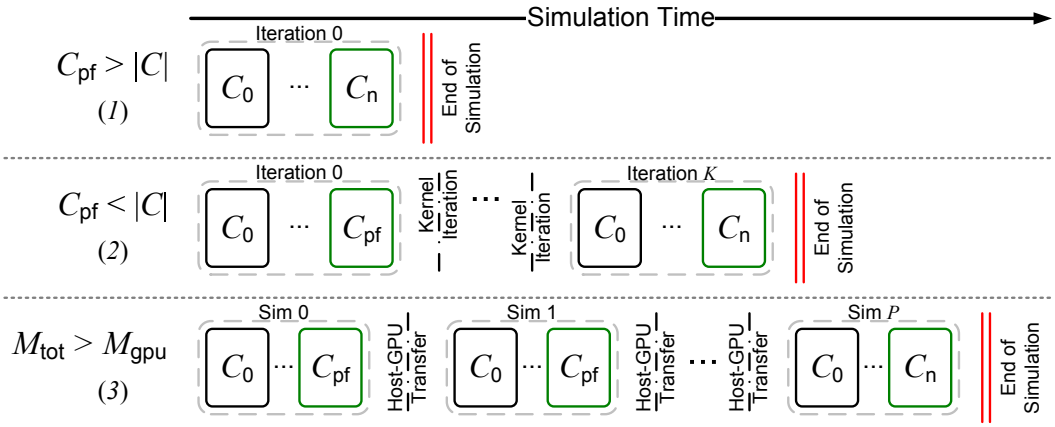


Figure 6.6: Three Simulation Scenarios

out requiring additional simulation (iteration) runs. This occurs when $C_{pf} \geq |MC|$, which is shown in Figure 6.6(a) whereby the entire set of MC can be simulated with one kernel iteration.

When the value of $wi_{\max}$ decreases due to the increased memory usage of the circuit stream data, then the value C_{pf} decreases. The host program creates a mapping of the circuit data of the form that is shown in Figure 6.6(b). This occurs when the circuit stream data consumes a significant amount of memory preventing the host program from allocating sufficient work-items. The kernel will then evaluate $K-1$ additional simulations, where K is defined as:

$$K = \left\lceil \frac{|MC|}{C_{pf}} \right\rceil \tag{6.6}$$

The additional kernel runs will add to the overall simulation time.

Figure 6.6(c) shows when the total memory M_{tot} required for simulation greatly exceeds the memory capacity on the GPU ($M_{\text{tot}} \gg M_{\text{gpu}}$). This causes the host program to partition the set of mutated circuits into P equal sets where each subset is simulated separately; however, this requires more than one data transfer from the host to the GPU. Thus, it is essential that the number of partitioned mutant streams be kept low.

6.6 Experimental Results

This section evaluates the run-time performance of μDV-GSIM. Table 6.2 lists the characteristics of the chosen designs. Two industrial designs and a PCI controller from OpenCores [109] were used for evaluating the performance of μDV-GSIM. The

Table 6.2: Charactistics of Each Design

Design	$\|A\|$	$\|T\|$	**Gates**
Indust. 1	67	803	4333
PCI	70	2121	19484
Indust. 2	82	1004	44956

first column shows the design name where the second column lists the number of assertions that were used ($|A|$). Each set of assertions was either readily provided with the design or subsequently written by analyzing the specifications. These designs were chosen because the assertions were capable of generating a large amount of tests, where the third column lists the size of test sets that were generated by the TG tool. The number of gates that is reported by the synthesis tool is shown in the fourth column.

The experimental platform that was employed to run the simulations consist of a host running on a six-core AMD processor at 3.2 GHz containing 16GB of memory. The GPU device is a GTX 560 Ti graphics card which contains 8 GPCs, each consisting of 48 processing cores and a core clock speed of 1.7 GHz. The available on-board memory is 1 GB with a maximum bandwidth of 128 GB/s. We used the NVIDIA CUDA SDK 4.2.1 using version 1.1 of the OpenCL language. The host's processor is responsible for serially simulating the set of mutated designs using an RTL simulator, whereas the GPU is used to run the circuit stream kernel for simulating multiple gate-level circuits concurrently.

To the best of the author's knowledge, there does not exist a mutation testing technique on GPUs for the context for assessing assertion quality. A direct comparison to other GPU-based circuit simulation tools is not feasible, due to the GPU hardware being more advanced than the ones indicated in the literature. Additionally, the fault simulation application for each proposed approach was used in the context of manufacturing testing, where μDV-GSIM is used for functional verification. Nevertheless, the memory savings gained from the proposed test data encoding techniques can be applied to the previous research approaches independently of the GPU architecture.

Achieving optimal performance with the μDV-GSIM simulator is dependent on the number of work-items that can be instantiated on the GPU. This implies that the simulation kernel should effectively use the available resources of every GPC. Each GPC inside the GPU platform consists of 48KB of local memory and 32,768 32-bit registers and is capable of executing 1536 work-items concurrently. To achieve 100% GPU work-item occupancy, each work-item limited to 32 bytes of local memory and

21 registers, allowing sufficient resources for other work-items to use within the GPC. The developed circuit stream OpenCL kernel consumes 19 registers and 8 bytes of local memory, thereby the proposed OpenCL implementation is capable in allocating 1536 work-items within a GPC. Since the GPU card contains 8 GPCs, then the simulation kernel can instantiate up to 12288 work-items. This implies that the maximum circuit parallelism factor, C_{pf} that can be achieved is 384 (when n_s is at the minimum required size); however, as explained in Section 6.5.2, the number of work-items that can be instantiated on the GPU ($wi_{\max}$) is dependent on the memory consumption of the circuit stream memory. This can affect the overall circuit parallelism factor.

Prior to executing the simulation kernel on the GPU, the host program computes the circuit parallelism factors when using two different assertion encoding schemes, namely Single Assertion Encoding (SAE) and the proposed Multiple Assertion Encoding (MAE). Each design was injected using 200 *synthesizable mutations* at the RTL level. This value was chosen conservatively because the the amount of mutated circuit data will force the host program to compute the different CPF factors, using the two different test encoding schemes. Additionally, it will allow μDV-GSIM to perform circuit simulation using the three different scenarios, as depicted in Figure 6.5, along with comparing the simulation performance.

The SAE encoding will have each work-item simulating one single set of tests that were generated from one assertion. The proposed MAE encoding generates 32-bit test packets containing multiple tests which can be simulated concurrently for each work-item. The intent is to show that these encoding schemes can affect the number of circuit streams within each circuit C_k, which directly affects the circuit parallelism factor. For each computed value of C_{pf}, their values of $wi_{\max}$ will be used for launching the OpenCL kernel on the GPU. From there, a comparison was made on the run-time between the two encoding schemes. Each design was simulated using the number of tests that are listed from Table 6.2 for ten execution runs, after which the run-times are averaged in order to ensure the measurements are consistent.

6.6.1 Circuit Parallelism Factor and Work-item Configuration

Table 6.3 lists the computed circuit parallelism factor for each of the encoding schemes. The first column (Design) lists the name of the benchmark while the second column (Encoding) list the encoding technique that was used for generating the circuit stream

Table 6.3: Computed Circuit Parallelism Factor (C_{pf}) for 200 Injected Mutations when using MAE and SAE Encoding

Design	**Encoding**	n_s	M_{CS} (MB)	M_{nets} (MB)	M_{tot} (MB)	wi_{max}	C_{pf}	**Sim. Scene.**
Indust. 1	SAE	96	158.7	203.1	361.8	12288	128	2
	MAE	32	52.9	203.1	256.0	12288	**384**	1
PCI	SAE	96	713.5	246.5	960.0	6144	64	2
	MAE	32	237.8	608.9	846.7	9216	**288**	1
Indust. 2	SAE	96	1097.6	351.2	1448.8	2048	32	3
	MAE	32	548.8	351.2	900.0	2048	**64**	3

data for 200 injected mutations. The number of circuit streams (n_s) per mutated circuit C_k is listed in the third column. The fourth and fifth columns list the memory consumed by the circuit stream data (M_{CS}) and the storing of the intermediate values (M_{nets}) by every work-item respectively. The total memory (M_{tot}) shown in the sixth column is the sum of M_{CS} and M_{nets}. The computed maximum number of work-items (wi_{max}) is listed in the seventh column. The value of M_{gpu} is set at 960MB, where the remaining memory can be used for storing software drivers used for communicating with the host. The computed circuit parallelism factor (C_{pf}) and the simulation scenarios (indicated as Sim. Scene.) are indicated in the eighth and ninth columns respectively. The reference for the simulation scenario labellings are defined in Figure 6.5.

As seen in the table, each design had an increase in the number of circuit streams (n_s) when using SAE encoding. The increase is necessary because SAE assigns one set of tests (T_s) for a work-item operating on one circuit stream. To assign the entire set of assertion-based tests within a circuit C_k using SAE, then the number of circuit streams (n_s) will increase by multiple factors of 32. For instance, every benchmark required n_s be at 96 in order to allocate their entire set of assertions. We see that the memory consumed by the circuit stream data for each design is larger when using SAE. When applying the proposed MAE encoding scheme, the number of circuit streams has decreased to its minimum size of 32 This is due to the fact that the host program was able to encode multiple assertion-based tests over the 32-bit computer word (i.e., wi_0 simulates assertion-based tests from assertion ϕ_0–ϕ_{31}). This helps in fitting the entire set of assertion-based tests without increasing the number of circuit streams in every circuit, C_k. Additionally, the memory consumption of the circuit stream data is reduced by 66% for each benchmark.

Every design experienced an improvement in its computed value of the circuit

Table 6.4: Performance Analysis of μDV-GSIM vs. RTL Simulator

Design	Average Simulation Run-times (s)			Speed-up vs RTL Sim (×)		Speed-up (×)
	MAE	SAE	RTL Sim	MAE	SAE	MAE vs SAE
Indust. 1	0.35	0.95	1.70	4.0	1.8	2.2
PCI	3.88	10.38	20.34	5.3	2.0	2.7
Indust. 2	5.11	12.79	54.23	10.6	4.2	2.5
			Average:	6.6	2.7	2.5

parallelism factor, C_{pf}. This was due to the reduction in the number of circuit streams that were assigned for each circuit, C_k, which allowed for more mutated circuits to be simulated concurrently. The improvement of the parallelism factor was also compounded by the increase in the maximum number of work-items, which was evident with the PCI design. The PCI benchmark had an increase in the number of work-items ($wi_{\max}$) from 6144 to 9216. This is due to the reduction in the memory consumption of the circuit stream data, which led to an improvement of the circuit parallelism factor of 288.

The Indust. 1 and Indust. 2 designs did not experience an increase to the number of allocated work-items. In Indust 1., the host program has allocated the maximum work-items that can be instantiated on the GPU platform. The circuit stream memory of Indust 2. is significantly large and exceeded the available memory on the GPU, which prevented the host program in allocating additional work-items. Despite the lack of improvement, the MAE encoding helped in improving the circuit parallelism factor. The Indust. 1 design was able to simulate its entire set of mutated circuits within one kernel iteration. In the Indust 2. design, the improvement in the circuit parallelism factor led to a reduction in the number of data transfers between the host and GPU. These improvements are reflective of the enhanced run-time performance of the μDV-GSIM tool, which is discussed in the next section.

6.6.2 Run-time Comparison with Different Assertion Encodings

Table 6.4 shows the average run-time comparisons of μDV-GSIM when employing the SAE and MAE assertion encoding schemes for each design. We also compared the performance with an RTL simulator which ran on the host's processor. The first column lists the names of the designs. Columns 2 through 4 lists the different run-times when using μDV-GSIM with the two encoding techniques, and the run-times

from the RTL simulator (shown as RTL Sim) tool. Columns 5 and 6 shows the achieved speed-ups against the RTL simulator when μDV-GSIM is employed with MAE and SAE encoding. Finally, column 7 shows the speed-up improvement when compared against the two encoding schemes.

The average run-times of the μDV-GSIM tool when employing the MAE encoding are visibly reduced when compared to the SAE encoding. Each design had an increase in the circuit parallelism factor which has led to an average speed-up of 6.6×, with a maximum of 10.6× speed-up, when compared to the performance of the RTL simulator tool. This is due to the fact that the improved circuit parallelism factor helped in reducing the number of kernel iterations (shown from Figure 6.6) that is required for simulating the entire set of circuits on the GPU. For instance, the Indust 1. and the PCI designs were able to simulate 200 circuits within one kernel iteration when using the MAE encoding scheme because their achieved circuit parallelism factors has exceeded the number of circuits required for simulation ($C_{pf} > |MC|$ where $|MC| = 200$). Additionally, the improved value of C_{pf} for the Indust. 2 design helped in the run-time on the GPU. This is because that SAE encoding generated a large amount of circuit data, which required additional transfers between the host and GPU. The MAE encoding technique reduce the memory consumption, thereby reducing the number of data transfers during simulation.

The proposed MAE encoding technique was able to reduce the circuit stream memory, which this was observed for designs containing a large set of assertion-based tests. As shown in the results, the gained memory savings led to an improvement in the simulation performance on the GPU. This is very encouraging for designs that are injected with a large quantity of mutations, for the purpose of rigourously measure the assertion quality.

6.7 Summary

Mutation testing is a key technique for analyzing the amount of coverage attained for a set of tests. Mutation testing can also be beneficial for measuring the quality of assertions that are used for verifying the correctness of a design. In this chapter, μDV-GSIM tool was introduced, a framework for assessing the quality of assertions using GPUs. The proposed approach attempts to exploit many levels of parallelism so that the GPU is able to simulate multiple mutated circuits with different assertion-based tests concurrently. The proposed Multiple Assertion Encoding generates a compact mapping of the mutated circuit and test data, while achieving additional simulation

performance of at least 2.5× compared to when every work-item is simulating a single set of assertion-based tests. The proposed techniques that was put forth allowed the handling of larger circuits that contain an abundance of assertions. This is beneficial in the quest for improving the assertion quality in design verification.

Chapter 7

Conclusions

This chapter summarizes the work that was presented in this thesis, along with some suggested future work.

7.1 Conclusions

Assertions are continuing to be a prominent force behind functional verification. The available assertion languages have enabled verification engineers to write assertions for checking the design's functional correctness using either static or dynamic verification methods. The expressive power of the assertion languages has led to the development of assertions describing complex behaviour to which the design must conform. This thesis has introduced a variety of methods and algorithms in leveraging the design behaviour expressed in assertions as a valuable source for generating tests. Additionally, this thesis shows how to assess the test quality from assertions, which can help in improving the assertion quality.

Dynamic verification is the most predominant method in functionally verifying the correctness of designs. When assertions are inserted into the design, they are treated as coverage points. The input stimuli is used for exerting the necessary conditions within the design in order to *cover* those assertions. As was shown in this thesis, using a *computable representation* of assertions for generating tests, such as Non-deterministic Finite Automata, can have more than one sequence of events that can cause an assertion to pass or fail. The proposed coverage metrics brought forth into this thesis have helped in gauging how thorough the assertion was evaluated. Additionally, the coverage metrics were then incorporated into the test generator, so that it gives the ability in generating tests without excluding any sequence of events within an assertion. It was shown that the developed coverage driven assertion-based

test generator produced additional tests that thoroughly evaluates the assertion, while improving the assertion coverage by as much as 70%. These tests can then be used in simulation, thereby performing an effective simulation for functionally verifying the correctness of the design.

Assertions are capable of generating a large amount of test data because the conditions that were defined in the assertion can use complex temporal sequences with large repetitions. Reducing the number of assertion-based tests required the development of a clustering algorithm for grouping assertions and sequences that contain a certain level of similarity. These similarities were exploited by the two developed test compaction algorithms. The first test compaction approach was to use the longest test sequence in the attempt of overlapping shorter sequences, thereby reducing the number of tests. The second improved approach relied on assigning multiple events from different assertions within a single functional test. This led to have one test in potentially satisfying (or falsifying) more than one assertion. Results showed the second test compaction algorithm yielded favourable test compaction, by as much as 85% and 70% improvement for compacting good and failing tests respectively, when compared to the initial attempt. Furthermore, it was empirically demonstrated that the proposed test compaction algorithms can be easily integrated with an assertion-based test generator developed by another research team. These test compaction algorithms visibly reduced the overall test set size by as much as 98% in some of the benchmarks, which can reduce the applied verification time.

GPU's are an ideal parallel platform for accelerating simulations of multiple mutated circuits; however, harnessing their raw compute power is dependent on the how the circuit data is organized and the amount of device memory that is needed for circuit simulation. This is crucial when performing mutation testing on a large set of mutated designs, for the purpose of assessing assertion quality. Additionally, the GPUs limited device memory can pose a challenge when replicating and generating data-parallel representations of mutated designs. Thus, careful duplication is needed, which led to the development of a novel data-parallel circuit data generation algorithm. The devised algorithm aims to efficiently use the memory resources that are available on the GPU. An innovative encoding scheme was devised, where the GPU's computer word length was leveraged for exploiting, mutant, gate, fault and test data parallelism factors. The developed data-parallel generation algorithm combined these parallelism factors, which enabled an efficient mapping of the multiple mutated and fault injected circuit data. The efficient encoding scheme has also enabled the development of a GPU simulation kernel in OpenCL, where each instance of a kernel is

simulating a unique mutated gate-level circuit. Furthermore, the devised mathematical formulations based on the memory usage of the circuit data, has helped the host program in computing an optimal work-item configuration for performing circuit simulation on the GPU. Results show that efficiently generating and duplicating circuit data has led to a decrease in memory consumption by as much as 80% in some of the benchmarks. This has led to an improvement in simulation run-time by as much as 5.4× on the GPU, and outperformed a commercial simulator by at least 95×.

The assertion-based test generation framework that was developed, along with the devised strategy for simulation multiple mutated circuits on the GPU, has allowed for the assessment of test quality from assertions using mutation testing. The assertion-based tests are used as input stimuli and are simulated over multiple circuits, where each contains a unique mutation (functional fault). Assertions are capable of generating a plethora of tests, which was shown to affect the generation of an efficient data-parallel representation of the mutated circuits, ultimately affecting simulation performance. This has led to a novel encoding scheme, where multiple assertion-based tests are encoded over the GPU's computer word, with the intent of reducing the memory usage of the mutated circuit data. It was observed that the 67% memory reduction improved simulation performance by as much as 10× on the GPU and 54× compared to an RTL simulator running on the host. This in turn helped in assessing the quality of assertions in negligible time, which is favourable for improving the quality of assertions.

As designs increase in complexity over time, dynamic simulation will undoubtedly continue to be the predominant method in verifying the design's functionality. The proposed algorithms and developed tools that were presented allow for the generation of effective test stimuli for dynamic simulation. Combining the effective assertion-based test generation strategy and a methodology for an accelerated assertion quality assessment will give verification engineers the means for improving the quality of tests, thereby performing functional verification using assertion-based dynamic verification effectively. It is imperative to ensure designs must be error free; however, the continuing rise of design complexity will no doubt bring significant functional verification challenges on the road to producing error-free designs.

7.2 Future Work

The research performed for generating assertion-based tests and using GPUs for accelerating mutation testing, can be further extended to future projects.

7.2.1 Assertion-based Test Generation

- The test generation strategy that was employed with *Airwolf-TG*, assumes that the given assertions reference the primary inputs and outputs. The assume-guarantee paradigm allowed for this assumption to be valid, thus generating tests is reduced to the automata traversal tasks, which is then addressed in a manner suitable for the types of NFA arising in hardware checkers. Assertions can reference signals inside the design, and this case the test generation by automata traversal alone does not result in the sequences of primary inputs. It would be suitable for using the sequences generated by *Airwolf-TG* with a model checking tool in order to assign the appropriate Boolean values at the primary inputs. Because of the known bounds on the sequence lengths, the bounded model checking approach (such as in [43]) is promising. Towards that end, incorporating model checkers such as NuSMV, together with MBAC and Airwolf-TG can yield a highly effective assertion-based test generation strategy.
- The assertion-based tests from *Airwolf-TG* can be used as test stimuli for functionally verifying designs that are synthesized on a reconfigurable fabric, such as a FPGA. Nowadays, digital designs have dedicated post-silicon debugging hardware for capturing certain behaviour of the design during operation. The assertion-based tests can be used for exerting the necessary conditions of the synthesized design on the FPGA. This can help to verify the implemented functionality and also assess how thorough the implemented design has satsified the specification during operation. Additionally, the compacted set of tests using assertion clustering can also be applied with designs having hardware assertion clusters as a post-silicon debug infrastructure. Clusters of hardware assertions, which were described in [66, 65], can benefit from the compacted assertion-based tests by quickly assessing how well the design was implemented.

7.2.2 GPU Accelerated Mutation Testing

- The work presented in Chapter 6 developed a strategy for accelerating simulations of multiple mutated designs using GPUs, for the context of mutation testing. Commercial and academic tools such as Certitude [110] and Laerte++ [111] analyze the design along with the set of assertions, for determining which mutation to inject and its location inside the RTL design. This will help the assertion-based tests to properly target the mutations that are necessary for *activating* and *propagating* the mutated result at the primary outputs. Since

those tools are also capable of injecting a large amount of mutations, then the proposed μDV-GSIM framework can help accelerate simulations of multiple mutated circuits containing effective mutations and report their coverages in negligible time.

- The GPU simulation algorithms that were presented operate on gate-level representations of digital designs. Even though, the developed algorithms have successfully exploited many levels of parallelism for designs provided at the gate-level, RTL simulations can further improve the simulation performance due to being at a higher level abstraction (such as in [74]). From there, faults can be injected at the RTL level without having the need for synthesizing each mutated design into its gate-level representation. This is beneficial for simulating multiple mutated circuits for assessing assertion quality without the need for synthesizing the mutated design.

7.2.3 Performance Comparison between CUDA and OpenCL

- The developed GPU simulation kernels shown in Chapters 5 and 6 were written in the OpenCL language. Another project is to explore the differences between CUDA and OpenCL in terms of register usage, local memory usage and run-time performance. This future work is beneficial in comparing the advantages and disadvantages between the two popular GPU kernel programming languages. Typically, designs are synthesized into their gate-level representations and companies perform gate-level logic or fault simulation in determining their correctness and the ability to detect manufacturing faults. This work would be valuable in helping to choose the appropriate GPU kernel language for accelerating logic and fault simulation algorithms, where gate-level simulations are the most time consuming during the design of VLSI circuits.

Bibliography

[1] H. Foster, *Post Silicon Debug Workshop – Design Automation Conference*, 2012. [Online]. Available: https://www.research.ibm.com/haifa/images/dac/Harry_2012-DAC-Post-Silicon-Workshop.pdf

[2] Y. Oddos, K. Morin-Allory, D. Borrione, M. Boulé, and Z. Zilic, "Mygen: automata-based on-line test generator for assertion-based verification," in *Proceedings of the 19th ACM Great Lakes symposium on VLSI (GLSVLSI)*. ACM, 2009, pp. 75–80.

[3] H. Foster, "Applied assertion-based verification: An industry perspective," *Foundations and Trends in Electronic Design Automation*, vol. 3, no. 1, pp. 1–95, 2009.

[4] H. Foster, A. Krolnik, and D. Lacey, *Assertion-based design*. Springer, 2004.

[5] O. Kupferman and M. Vardi, "Vacuity detection in temporal model checking," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 4, no. 2, pp. 224–233, 2003.

[6] M. Boulé and Z. Zilic, *Generating Hardware Assertion Checkers: For Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring*. Springer Publishing Company, Incorporated, 2008.

[7] J. G. Tong, M. Boulé, and Z. Zilic, "Defining and providing coverage for assertion-based dynamic verification," *Journal of Electronic Testing*, vol. 26, no. 2, pp. 211–225, 2010.

[8] J. G. Tong, M. Boulé, and Z. Zilic, "Airwolf-TG: A test generator for assertion-based dynamic verification," in *Proceedings of the 2009 International High Level Design Validation and Test Workshop (HLDVT)*. IEEE, 2009, pp. 106–113.

[9] *ARM AMBA 3 Specification and Assertions*, ARM Holdings plc, 2012. [Online]. Available: http://www.arm.com/producs/solutions/axi_spec.html

[10] N. Een and N. Sörensson, "The minisat page," *Research Web Page. Chalmers University, Sweden (March 2007) http://minisat.se*, 2006.

[11] J. G. Tong, M. Boulé, and Z. Zilic, "Test compaction techniques for assertion-based test generation," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 19, no. 1, p. 9, 2013.

[12] J. G. Tong, M. Boulé, and Z. Zilic, "Assertion clustering for compacted test sequence generation," in *Proceedings of the 13th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2012, pp. 694–701.

[13] J. G. Tong, M. Boulé, and Z. Zilic, "Efficient data encoding for improving fault simulation performance on gpus," in *Proceedings of the 4th International Symposium on Electronic System Design (ISED)*. IEEE, 2013, p. (In Press).

[14] J. G. Tong, M. Boulé, and Z. Zilic, "Mu-gsim: A mutation testing simulator on gpus," in *Proceedings of the 5th Asia Symposium on Quality Electronic Design (ASQED)*. IEEE, 2013, pp. 302–311.

[15] C. Spear, *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer, 2008.

[16] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2008.

[17] H. B. Carter and S. G. Hemmady, *Metric Driven Design Verification: An Engineer's and Executive's Guide to First Pass Success*. Springer, 2007.

[18] *Property Specification Language Reference Manual*, Accelera, June 2004. [Online]. Available: http://www.eda.org/vfv/docs/PSL-v1.1.pdf

[19] *1800-2005 - IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language*, IEEE, 2005. [Online]. Available: http://standards.ieee.org/findstds/standard/1800-2005.html

[20] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model checking*. MIT press, 1999.

[21] P. Dasgupta, *A roadmap for formal property verification*. Springer, 2006.

[22] B. Pal, A. Banerjee, A. Sinha, and P. Dasgupta, "Accelerating assertion coverage with adaptive testbenches," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 5, pp. 967–972, 2008.

[23] J. Yuan, C. Pixley, and A. Aziz, *Constraint-based verification*. Springer, 2006.

[24] D. Borrione, K. Morin-Allory, and Y. Oddos, "Property-based dynamic verification and test," in *Design Technology for Heterogeneous Embedded Systems*. Springer, 2012, pp. 157–176.

[25] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[26] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *Proceedings of the 15th international conference on Software Engineering*. IEEE Computer Society Press, 1993, pp. 100–107.

[27] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[28] C. Nvidia, *Compute unified device architecture programming guide*, 2013.

[29] A. Advanced Micro Devices, *AMD Accelerated Parallel Processing*, 2013.

[30] *NVIDIA OpenCL Programming Guide*, 2011.

[31] J. Croix, K. Gulati, and S. Khatri, "Using gpus to accelerate cad algorithms," *IEEE Design and Test*, vol. 30, pp. 8–16, 2013.

[32] P. Banerjee, *Parallel algorithms for VLSI computer-aided design*. Prentice-Hall, Inc., 1994.

[33] K. Gulati and S. Khatri, "Towards acceleration of fault simulation using graphics processing units," in *Proceedings of the 45th annual Design Automation Conference (DAC)*. ACM, 2008, pp. 822–827.

[34] M. B. Amin and B. Vinnakota, "Workload distribution in fault simulation," *Journal of Electronic Testing*, vol. 10, no. 3, pp. 277–282, 1997.

[35] N. Manjikian and W. M. Loucks, "High performance parallel logic simulations on a network of workstations," in *Proceedings of the seventh workshop on Parallel and distributed simulation*, vol. 23, no. 1. ACM, 1993.

[36] S.-E. Tai and D. Bhattacharya, "Pipelined fault simulation on parallel machines using the circuit flow graph," in *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*. IEEE, 1993, pp. 564–567.

[37] N. Ishiura, M. Ito, and S. Yajima, "Dynamic two-dimensional parallel simulation technique for high-speed fault simulation on a vector processor," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 8, pp. 868–875, 1990.

[38] M. Amin and B. Vinnakota, "Data parallel fault simulation," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 2, pp. 183–190, 1999.

[39] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU architecture," *IEEE Micro*, vol. 31, no. 2, pp. 50–59, 2011.

[40] *The OpenCL Specification Version 1.1 – Khronos OpenCL Working Group*, 2011.

[41] Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, "Using Fermi architecture knowledge to speed up CUDA and OpenCL programs," in *Proceedings of the International Symposium on Parallel and Distributed Processing with Applications (ISPA)*. IEEE, 2012, pp. 617–624.

[42] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause *et al.*, "Using formal specifications to support testing," *ACM Computing Surveys (CSUR)*, vol. 41, no. 2, p. 9, 2009.

[43] H.-M. Koo and P. Mishra, "Test generation using sat-based bounded model checking for validation of pipelined processors," in *Proceedings of the 16th ACM Great Lakes symposium on VLSI (GLSVLSI)*. ACM, 2006, pp. 362–365.

[44] K. Shimizu and D. L. Dill, "Deriving a simulation input generator and a coverage metric from a formal specification," in *Proceedings of the 39th annual Design Automation Conference*. ACM, 2002, pp. 801–806.

[45] J. R. Calamé, *Specification-based test generation with TGV*. Centrum voor Wiskunde en Informatica, 2005.

[46] K. T. Cheng and A. Krishnakumar, "Automatic functional test generation using the extended finite state machine model," in *Proceedings of the 30th international Design Automation Conference (DAC)*. ACM, 1993, pp. 86–91.

[47] G. Di Guglielmo, F. Fummi, G. Pravadelli, S. Soffia, and M. Roveri, "Semi-formal functional verification by efsm traversing via nusmv," in *Proceedings of the 2010 International High Level Design Validation and Test Workshop (HLDVT)*. IEEE, 2010, pp. 58–65.

[48] V. Boppana, S. P. Rajan, K. Takayama, and M. Fujita, "Model checking based on sequential atpg," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*. Springer, 1999, pp. 418–430.

[49] J. Dworak, K. Nepal, N. Alves, Y. Shi, N. Imbriglia, and R. Iris Bahar, "Using implications to choose tests through suspect fault identification," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 18, no. 1, p. 14, 2012.

[50] M. Bushnell and V. Agrawal, *Essentials of electronic testing for digital, memory, and mixed-signal VLSI circuits*. Springer, 2000, vol. 17.

[51] M. S. Hsiao, E. M. Rudnick, and J. H. Patel, "Fast algorithms for static compaction of sequential circuit test vectors." in *Proceedings of the VLSI Test Symposium (VTS)*. IEEE, 1997, pp. 188–195.

[52] I. Pomeranz and S. M. Reddy, "Dynamic test compaction for synchronous sequential circuits using static compaction techniques," in *Proceedings of the Annual Symposium on Fault Tolerant Computing*. IEEE, 1996, pp. 53–61.

[53] E. M. Rudnick and J. H. Patel, "Efficient techniques for dynamic test sequence compaction," *IEEE Transactions on Computers*, vol. 48, no. 3, pp. 323–330, 1999.

[54] I. Pomeranz and S. Reddy, "On test generation with test vector improvement," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 3, pp. 502–506, 2010.

[55] S. Neophytou and M. Michael, "Test set generation with a large number of unspecified bits using static and dynamic techniques," *IEEE Transactions on Computers*, vol. 59, no. 3, pp. 301–316, 2010.

[56] A. H. El-Maleh and Y. E. Osais, "Test vector decomposition-based static compaction algorithms for combinational circuits," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 8, no. 4, pp. 430–459, 2003.

[57] I. Pomeranz and S. M. Reddy, "Forward-looking fault simulation for improved static compaction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 10, pp. 1262–1265, 2001.

[58] P. Drineas and Y. Makris, "Independent test sequence compaction through integer programming," in *Proceedings of the 21st International Conference on Computer Design (ICCD)*. IEEE, 2003, pp. 380–386.

[59] M. Dimopoulos and P. Linardis, "Efficient static compaction of test sequence sets through the application of set covering techniques," in *Proceedings of the 2004 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, vol. 1. IEEE, 2004, pp. 194–199.

[60] Pomeranz, Irith and Reddy, Sudhakar M, "Static test compaction for full-scan circuits based on combinational test sets and non-scan sequential test sequences," in *Proceedings. 16th International Conference on VLSI Design.* IEEE, 2003, pp. 335–340.

[61] I. Pomeranz and S. M. Reddy, "Static test compaction for scan-based designs to reduce test application time," *Journal of Electronic Testing*, vol. 16, no. 5, pp. 541–552, 2000.

[62] Y. Cho, I. Pomeranz, and S. M. Reddy, "On reducing test application time for scan circuits using limited scan operations and transfer sequences," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 10, pp. 1594–1605, 2005.

[63] I. Pomeranz and S. M. Reddy, "Autoscan: a scan design without external scan inputs or outputs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 13, no. 9, pp. 1087–1095, 2005.

[64] M. Chen and P. Mishra, "Functional test generation using efficient property clustering and learning techniques," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 3, pp. 396–404, 2010.

[65] M. Neishaburi and Z. Zilic, "Enabling efficient post-silicon debug by clustering of hardware-assertions," in *Proceedings of the 2010 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2010, pp. 985–988.

[66] M. Boulé, J.-S. Chenard, and Z. Zilic, "Assertion checkers in verification, silicon debug and in-field diagnosis," in *Proceedings of the 8th International Symposium on Quality Electronic Design*. IEEE, 2007, pp. 613–620.

[67] M. Gao and K.-T. Cheng, "A case study of time-multiplexed assertion checking for post-silicon debugging," in *Proceedings of the 2010 High Level Design Validation and Test Workshop (HLDVT)*. IEEE, 2010, pp. 90–96.

[68] H.-M. Koo and P. Mishra, "Specification-based compaction of directed tests for functional validation of pipelined processors," in *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*. ACM, 2008, pp. 137–142.

[69] M. Li and M. Hsiao, "3-D parallel fault simulation with GPGPU," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 10, pp. 1545–1555, 2011.

[70] M. Kochte, M. Schaal, H. Wunderlich, and C. Zoellin, "Efficient fault simulation on many-core processors," in *Proceedings of the 47th Design Automation Conference (DAC)*. IEEE, 2010, pp. 380–385.

[71] D. Chatterjee, A. Deorio, and V. Bertacco, "Gate-level simulation with GPU computing," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 16, no. 3, p. 30, 2011.

[72] A. Sen, B. Aksanli, M. Bozkurt, and M. Mert, "Parallel cycle based logic simulation using graphics processing units," in *Proceedings of the Ninth International Symposium on Parallel and Distributed Computing (ISPDC)*. IEEE, 2010, pp. 71–78.

[73] A. Sen, B. Aksanli, and M. Bozkurt, "Speeding up cycle based logic simulation using graphics processing units," *International Journal of Parallel Programming*, vol. 39, no. 5, pp. 639–661, 2011.

[74] N. Bombieri, F. Fummi, and V. Guarnieri, "FAST-GP: An RTL functional verification framework based on fault simulation on GP-GPUs," in *Proceedings*

of the Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2012, pp. 562–565.

[75] N. Bombieri, S. Vinco, V. Bertacco, and D. Chatterjee, "SystemC simulation on GP-GPUs: CUDA vs. OpenCL," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES)*. ACM, 2012, pp. 343–352.

[76] S. Holst, E. Schneider, and H.-J. Wunderlich, "Scan Test Power Simulation on GPGPUs," in *Proceedings of the 21st Asian Test Symposium (ATS)*. IEEE, 2012, pp. 155–160.

[77] M. Li, K. Gent, and M. S. Hsiao, "Utilizing gpgpus for design validation with a modified ant colony optimization," in *Proceedings of the 2011 International High Level Design Validation and Test Workshop (HLDVT)*. IEEE, 2011, pp. 128–135.

[78] M. Li and M. S. Hsiao, "Rag: an efficient reliability analysis of logic circuits on graphics processing units," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. EDA Consortium, 2012, pp. 316–319.

[79] J. Offutt, P. Ammann, and L. Liu, "Mutation testing implements grammar-based testing," in *Proceedings of the Second Workshop on Mutation Analysis*. IEEE, 2006, pp. 12–12.

[80] M. Sousa and A. Sen, "Generation of TLM testbenches using mutation testing," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES)*. ACM, 2012, pp. 323–332.

[81] N. Bombieri, F. Fummi, V. Guarnieri, and G. Pravadelli, "Testbench qualification of SystemC TLM protocols through Mutation Analysis," *IEEE Transactions on Computers*, p. (In Press), 2012.

[82] P. Behnam, B. Alizadeh, Z. Navabi, and M. Fujita, "Mutation based debugging technique with auto-correction mechanism for RTL designs," in *8th International Workshop on Silicon Debug and Diagnosis*. IEEE, 2012, p. In Press.

[83] S. Mirzaeian, F. Zheng, and K.-T. Cheng, "RTL error diagnosis using a word-level SAT-solver," in *Proceedings on the 2008 International Test Conference (ITC)*. IEEE, 2008, pp. 1–8.

[84] M. S. Abadir, J. Ferguson, and T. E. Kirkland, "Logic design verification via test generation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 7, no. 1, pp. 138–148, 1988.

[85] S. Katz, O. Grumberg, and D. Geist, ""have i written enough properties?"-a method of comparison between specification and implementation," in *Correct hardware design and verification methods*. Springer, 1999, pp. 280–297.

[86] O. Kupferman, W. Li, and S. A. Seshia, "A theory of mutations with applications to vacuity, coverage, and fault tolerance," in *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*. IEEE Press, 2008, p. 25.

[87] G. Fraser and F. Wotawa, "Using model-checkers for mutation-based test-case generation, coverage analysis and specification analysis," in *Proceedings of the International Conferenc eon Software Engineering Advances*. IEEE, 2006, pp. 16–16.

[88] S. Ben-David, D. Fisman, and S. Ruah, "Temporal antecedent failure: Refining vacuity," in *CONCUR 2007–Concurrency Theory*. Springer, 2007, pp. 492–506.

[89] R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Y. Vardi, "Enhanced vacuity detection in linear temporal logic," in *Proceedings of the International Conference Computer Aided Verification*. Springer, 2003, pp. 368–380.

[90] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, "Efficient detection of vacuity in actl formulas," in *Proceedings of the International Conference on Computer Aided Verification*, 1997, pp. 279–290.

[91] V. Singhal and P. Aggarwal, "Using coverage to deploy formal verification in a simulation world," in *Proceedings of the International Conference on Computer Aided Verification*. Springer, 2011, pp. 44–49.

[92] L. Di Guglielmo, F. Fummi, and G. Pravadelli, "Vacuity analysis by fault simulation," in *Proceedings of the International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*. IEEE, 2008, pp. 27–36.

[93] G. Di Guglielmo, F. Fummi, and G. Pravadelli, "The role of mutation analysis for property qualification," in *Proceedings of the International Conference on*

Formal Methods and Models for Co-Design (MEMOCODE). IEEE, 2009, pp. 28–35.

[94] A. Banerjee, B. Pal, C. Kamarapu, P. Dasgupta, P. Chakrabarti, and M. Jha, "Assertion based verification: have i written enough properties?" in *Proceedings of the First India Annual Conference (INDICON)*. IEEE, 2004, pp. 363–367.

[95] G. Di Guglielmo, F. Fummi, M. Hampton, G. Pravadelli, and F. Stefanni, "The role of parallel simulation in functional verification," in *Proceedings of the International High Level Design Validation and Test Workshop (HLDVT)*. IEEE, 2008, pp. 117–124.

[96] D. A. Mathaikutty, S. Ahuja, A. Dingankar, and S. Shukla, "Model-driven test generation for system level validation," in *Proceedings of the 2007 High Level Design Validation and Test Workshop (HLDVT)*. IEEE, 2007, pp. 83–90.

[97] D. Chatterjee and V. Bertacco, "Activity-based refinement for abstraction-guided simulation," in *Proceedings of the High Level Design Validation and Test Workshop (HLDVT)*. IEEE, 2009, pp. 146–153.

[98] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, "Efficient detection of vacuity in temporal model checking," *Formal Methods in System Design*, vol. 18, no. 2, pp. 141–163, 2001.

[99] T. Ball and O. Kupferman, "Vacuity in testing," in *Tests and Proofs*. Springer, 2008, pp. 4–17.

[100] T. L. Anderson, "Coverage is the heart of verification," *EETimes.*, 2005. [Online]. Available: http://www.eetimes.com/document.aspdoc_id=1217979

[101] L. Bening and H. Foster, "Principles of verifiable rtl design," *Principles of Verifiable RTL Design: A Functional Coding Style Supporting Verification Processes in Verilog*, pp. 223–230, 2002.

[102] J. Sordoillet and S. Davey, "Integrated, comprehensive assertion-based coverage," in *EDA Tech Forum*, vol. 3, no. 1, 2006, pp. 22–25.

[103] M. Boulé and Z. Zilic, "Efficient automata-based assertion-checker synthesis of seres for hardware emulation," in *Proceedings of the 2007 Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2007, pp. 324–329.

[104] M. Boulé, J.-S. Chenard, and Z. Zilic, "Debug enhancements in assertion-checker generation," *IET Computers & Digital Techniques, IET*, vol. 1, no. 6, pp. 669–677, 2007.

[105] S. Vijayaraghavan and M. Ramanathan, *A Practical Guide for SystemVerilog Assertions*. Springer, 2005.

[106] J. Croix and S. Khatri, "Introduction to GPU programming for EDA," in *Proceedings of the 2009 International Conference on Computer-Aided Design (ICCAD)*. ACM, 2009, pp. 276–280.

[107] *ITC'99 Benchmarks*. [Online]. Available: http://www.cad.polito.it/downloads/tools/itc99.html

[108] *Virginia Tech VLSI for Telecommunications*. [Online]. Available: http://www.vtvt.ece.vt.edu/vlsidesign/cadtools.php

[109] *OpenCores*. [Online]. Available: http://opencores.org

[110] M. Hampton and S. Petithomme, "Leveraging a commercial mutation analysis tool for research," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*. IEEE, 2007, pp. 203–209.

[111] A. Fin and F. Fummi, "Laerte++: an object oriented high-level tpg for systemc designs." in *FDL*, vol. 3. Springer, 2003, pp. 105–117.

[112] C. Eisner and D. Fisman, *A Practical Introduction to PSL (Series on Integrated Circuits and Systems)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[113] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation, 2nd*. Addison-Wesley, 2001.

[114] G. D. Hachtel and F. Somenzi, *Logic synthesis and verification algorithms*. Kluwer academic publishers, 2000.

[115] A. Piziali, *Functional verification coverage measurement and analysis*. Springer, 2004.

[116] C. Baier, J.-P. Katoen *et al.*, *Principles of model checking*. MIT press Cambridge, 2008, vol. 26202649.

[117] Y. Oddos, K. Morin-Allory, and D. Borrione, "On-line test vector generation from temporal constraints written in psl," in *Proceedings of the 2006 International Conference on Very Large Scale Integration*. IEEE, 2006, pp. 397–402.

[118] X. Cheng and M. S. Hsiao, "Simulation-directed invariant mining for software verification," in *Proceedings of the conference on Design, automation and test in Europe (DATE)*. ACM, 2008, pp. 682–687.

[119] H. Lee and D. Ha, "An efficient, forward fault simulation algorithm based on the parallel pattern single fault propagation," in *Proceedings of the 1991 International Test Conference (ITC)*. IEEE, 1991, p. 946.

[120] D. Chatterjee, A. DeOrio, and V. Bertacco, "Event-driven gate-level simulation with GP-GPUs," in *Proceedings of the 46th Annual Design Automation Conference (DAC)*. ACM, 2009, pp. 557–562.

[121] G. J. Holzmann, "The model checker spin," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[122] J. Brzozowski, "Canonical regular expressions and minimal state graphs for definite events," *Mathematical theory of Automata*, vol. 12, pp. 529–561, 1962.

[123] M. Boulé and Z. Zilic, "Automata-based assertion-checker synthesis of PSL properties," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 13, no. 1, pp. 1–21, 2008.

[124] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking." *Handbook of Satisfiability*, vol. 185, pp. 457–481, 2009.