Efficiency and Applications of SAT-Based Test Pattern Generation

— Complex fault models and optimisation problems —

Dissertation zur Erlangung des Doktorgrades (Dr. rer. nat.)
der Technischen Fakultät
der Albert-Ludwigs-Universität
in Freiburg im Breisgau

vorgelegt von

Alexander Czutro,
geb. Yánez-Trujillo

im Juni 2013

Alexander Czutro - Albert-Ludwigs-Univerſität Freiburg

Dekan der Fakultät:

Prof. Dr. Yiannos Manoli

Gutachter:

Prof. Dr. Bernd Becker
Albert-Ludwigs-Universität Freiburg
Freiburg im Breisgau

Prof. Sudhakar M. Reddy
University of Iowa
Iowa City, IA, USA

Datum der Promotion:

12. August 2013

Acknowledgements

First of all, thanks to the adviser of this thesis, Prof. Dr. Bernd Becker, for all the support, help and encouragement I have received from him over the course of many years, and also for having introduced me to the world of scientific research at a very early stage of my undergraduate studies, which has significantly shaped my life in a positive way. And in particular, for giving me the opportunity to present my work at numerous symposia and workshops, thus allowing me to obtain valuable feedback from a large, international community of researchers.

Very special thanks to Prof. Sudhakar M. Reddy (University of Iowa) for the helpful tips that he provided in multiple occasions, and in general, for his ever-present willingness to share his vast knowledge.

My work would not have been possible without the help of fellow doctoral students in Freiburg: Dr. Matthew Lewis, Dr. Piet Engelke and Dr. Tobias Schubert, who provided the fault simulator and the SAT solvers that I integrated into my ATPG framework Tiguan. In particular, thanks to Matt and Piet, who spent a lot of time optimising their tools and implementing customisations during the initial stage of my doctoral research. Also, thanks to Dr. Stefan Hillebrecht and to Mr. Matthias Sauer for numerous technical discussions.

Thanks also to several people for their cooperation in different works: Prof. Sandip Kundu (University of Massachusetts), Prof. Michel Renovell and Prof. Mariane Comte (Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier), Prof. Dr. Sybille Hellebrand (University of Paderborn), Prof. Dr. Hans-Joachim Wunderlich (University of Stuttgart), Dr. Wolfgang Vermeiren (Fraunhofer Institute for Integrated Circuits, Dresden); and to Mr. Jürgen Schlöffel (Mentor Graphics, formerly NXP Semiconductors GmbH, Hamburg) for providing industrial benchmarks.

Really big thanks to Mr. Peter Winterer, Prof. Becker's system administrator, for taking care of a great deal of computing infrastructure that we students, immersed in the daily grind of doctoral research, sometimes take for granted; and for always being an attentive listener and an obliging colleague.

Finally, my greatest debts. To my former Freiburg colleague, Prof. Dr. Ilia Polian (University of Passau), for his guidance and patience teaching me innumerable skills over a period of ten years; and for proofreading this manuscript at such short notice. *And to my wife Kinga, whom I owe everything else in life.*

Alexander Czutro

Abstract

Modern technologies have enabled the semiconductor industry to enter a new era of integrated-circuit manufacturing. Modern ICs are not only smaller and significantly more high-performing than they used to be only a few years ago; they are also considerably more energy-efficient thanks to the use of new materials with convenient electric properties. However, the use of new materials is also making the fabrication process more difficult to control, and the new chips are more prone to defects. In consequence, the role of fault models that allow to describe complex forms of faulty behaviour is becoming increasingly important in hardware test and diagnosis.

Without doubt, automatic test pattern generation (ATPG) is the most important test task. ATPG algorithms need to be not only run-time-efficient and to produce compact test sets, given the large number of faults that need to be targeted in multi-billion-transistor ICs; they also need to keep pace with the development of new mechanisms for the description of faulty behaviour.

Traditionally, ATPG algorithms used in industrial applications are structural, i.e. their reasoning is based on the circuit's structure. However, SAT-based algorithms, i.e. methods that map the ATPG problem to the problem of Boolean satisfiability (SAT), have recently started to gain relevance because they perform better than structural methods on important classes of faults.

This doctoral thesis covers the work on SAT-based test pattern generation performed by the thesis's author between 2008 and 2012. It presents the SAT-based ATPG tool Tiguan and explains in detail all important aspects that were considered in order to make Tiguan a highly efficient test pattern generator capable of calculating provably optimal solutions for complex ATPG problems. The most important contributions of the work presented in this thesis can be summarised as follows:

- The general run-time efficiency of SAT-based ATPG was increased through intelligent mapping of the ATPG problem to SAT, through the optimal utilisation of multiple computing cores, and through the employment of advanced SAT solving techniques.

- Dynamic compaction was integrated into SAT-based ATPG. This allowed TIGUAN to test all stuck-at faults in ISCAS and ITC'99 circuits using less test patterns than a commercial, structural tool. Regarding the application to industrial circuits, the compaction efficiency gap between SAT-based and structural ATPG was significantly diminished.
- Generic fault models were defined which allow to represent complex defect behaviour. In addition, a flexible SAT-based framework for the generation of provably optimal test patterns for complex fault models was implemented. The applicability of the framework was illustrated by several example applications whose replication using structural methods is not trivial.
- The performed research and the created software code base opened the path to advanced research in small-delay test, variability and fault tolerance.

Each chapter of the thesis focuses on one key aspect, provides a thorough motivation for the work on that aspect, discusses all relevant algorithmic details, presents and analyses extensive experimental results, and points out important directions for future research. To conclude, the thesis reviews selected works by other authors which have benefited from TIGUAN's development.

Finally, after a brief summary of the presented topics, the thesis closes with a discussion of the role that SAT-based ATPG is expected to play in future industrial applications.

Zusammenfassung

Dank moderner Technologien befindet sich die Halbleiterindustrie in einer neuen Ära der Herstellung von integrierten Schaltungen (*integrated circuits* — IC). Moderne ICs sind nicht nur kleiner und deutlich performanter als vor nur wenigen Jahren; sie sind dank des Einsatzes neuer Werkstoffe mit günstigen elektrischen Eigenschaften auch wesentlich energieeffizienter geworden. Allerdings ist das Herstellungsverfahren durch die Verwendung von neuen Werkstoffen auch schwieriger steuerbar geworden, was zur Folge hat, dass die neuen Chips defektanfälliger sind. Vor diesem Hintergrund ist in der Hardwaretest-Forschung insbesondere die Rolle von Fehlermodellen, mit denen komplexe Formen von Fehlverhalten beschrieben werden können, zunehmend wichtiger geworden.

Ohne Zweifel ist die automatische Testmustererzeugung (*automatic test pattern generation* — ATPG) die wichtigste Testaufgabe. ATPG-Algorithmen müssen nicht nur laufzeiteffizient sein und kompakte Testmengen erzeugen, angesichts der großen Zahl von Fehlern, die in ICs mit mittlerweile mehreren Milliarden Transistoren betrachten werden müssen. Sie müssen auch mit der Entwicklung neuer Verfahren für die Beschreibung von Fehlverhalten Schritt halten.

Traditionell sind die ATPG-Algorithmen, die in industriellen Anwendungen eingesetzt werden, strukturell. Das heißt, ihr Suchverhalten wird von der Schaltungsstruktur diktiert. Allerdings haben vor relativ kurzer Zeit auch SAT-basierte Algorithmen angefangen, an Bedeutung zuzunehmen, da sie bei Anwendung auf bestimmte, wichtige Klassen von Fehlern eine bessere Leistung als strukturelle Methoden erbringen. SAT-basiert bedeutet, dass diese Methoden das ATPG-Problem auf das Problem der Boolschen Erfüllbarkeit (*Boolean satisfiability* — SAT) reduzieren.

Diese Dissertation umfasst die Arbeit, die der Autor im Forschungsbereich der SAT-basierten Testmustererzeugung zwischen 2008 und 2012 geleistet hat. Die Arbeit stellt das SAT-basierte ATPG-Werkzeug Tiguan vor und erklärt im Detail alle wichtigen Aspekte, die berücksichtigt werden mussten, um aus Tiguan ein hocheffizientes Testmustererzeugungswerkzeug zu machen, das in der Lage ist, beweisbar optimale Lösungen für komplexe ATPG-Probleme zu berechnen. Die wichtigsten Beiträge der in dieser Dissertation vorgestellten Arbeit können wie folgt zusammengefasst werden:

- Die allgemeine Laufzeiteffizienz von SAT-basiertem ATPG wurde durch die geeignete Abbildung des ATPG-Problems auf SAT, durch die optimale Nutzung mehrerer Rechenkerne, und durch den Einsatz von fortgeschrittenen SAT-Techniken verbessert.
- Ein Verfahren zur dynamischen Kompaktierung wurde in den SAT-basierten ATPG-Algorithmus integriert. Dies ermöglicht TIGUAN, alle Stuck-at-Fehler in ISCAS- und ITC'99-Schaltungen mit weniger Testmustern zu testen als ein kommerzielles strukturelles ATPG-Werkzeug. Was die Anwendung auf industrielle Schaltungen betrifft, so wurde die Kluft, die es zwischen SAT-basierten und strukturellen Methoden hinsichtlich der Testmengenkompaktheit gab, deutlich verkleinert.
- Generische Fehlermodelle wurden definiert, mit deren Hilfe sich komplexes Defektverhalten darstellen lässt. Darüber hinaus ist ein flexibles SAT-basiertes Werkzeug entstanden, mit dem die Erzeugung von beweisbar optimalen Testmustern für komplexe Fehlermodelle möglich ist. Die Anwendbarkeit des Konzeptes wurde anhand von mehreren Beispielanwendungen bewiesen, die sich mit strukturellen Methoden schwer realisieren lassen.
- Die durchgeführte Forschung und die entstandene Software-Codebasis eröffneten den Weg für weitere Forschung über Verzögerungsfehler, Variabilität und Fehlertoleranz.

Jedes Kapitel der Dissertation konzentriert sich auf einen zentralen Aspekt. Es bietet dabei eine gründliche Motivation für die realisierte Arbeit, erklärt alle relevanten algorithmischen Details, präsentiert und analysiert umfangreiche experimentelle Ergebnisse und erarbeitet Ideen für zukünftige Forschung. Am Ende der Arbeit werden ausgewählte Werke von anderen Autoren kurz vorgestellt, die von der Entwicklung von TIGUAN profitiert haben.

Nach einer kurzen Zusammenfassung der vorgestellten Themen widmet sich die Arbeit schließlich einer Diskussion über die Rolle, die man von der SAT-basierten Testmustererzeugung in zukünftigen industriellen Anwendungen erwarten darf.

Contents

List of Algorithms

List of Figures

List of Tables

If he had a needle to find in a haystack, he would proceed at once with the diligence of the bee to examine straw after straw until he found the object of his search…

I was a sorry witness of such doings, knowing that a little theory and calculation would have saved him ninety per cent of his labour.

— Nikola Tesla

1

Preface

During the last decade, modern semiconductor technologies have progressed to a level that allows the fabrication of high-performance *integrated circuits* (IC) that can be deployed into a wide variety of devices of daily use, like mobile phones, "smart watches" and even door locks. In large, this development has been made possible by the increased ability to miniaturise circuit components. In *CMOS* (Complementary Metal-Oxide-Semiconductor [254]) designs, feature sizes of 45 nanometres and less have become common. But newer technologies have also managed to deal with other important issues. The *HKMG* (High-*k*/Metal Gate [105, 35]) technology, for example, is a CMOS variant that replaces silicon dioxide with materials with a higher permittivity, which results in ICs with considerably higher energy efficiency and less heat dissipation. For instance, the heat dissipation of the Exynos 4 processor, a 32nm HKMG chip with four computing cores that can be operated at 1.6 GHz, is so low that the IC is being used in Samsung's newest high-end mobile phones [6]. In comparison, an Intel Pentium 4 (single-core) CPU deployed in desktop PCs in the year 2000 could be operated at a maximum speed of 1.5 GHz and could reach temperatures around 100°C [4].

The downside of these technologies, however, is that the fabrication process is becoming increasingly difficult to control, as the new materials have different properties. In consequence, new chips are more prone to defects. The 2011 International Technology Roadmap for Semiconductors lists the emergence of new technologies as one of the three key driver areas that will shape the future development of test methods and test equipment [10]. Hardware test is one of the most important tasks in the semiconductor production process. And its relevance is not characterised only by the necessity to identify faulty devices. Test and diagnosis are also crucial in that they produce feedback without which the semiconductor industry would not be able to improve their manufacturing processes.

In principle, the *test of a digital circuit* consists of an experiment in which a set of value combinations (*test patterns*) are applied to each manufactured circuit. If the values produced by a *circuit under test* (CUT) differ from the expected responses at any time, then the CUT is known to be defective. In addition, further analysis known as *diagnosis* can be performed on each CUT that fails the test in order to determine the cause of failure.

Expressed in this form, the test experiment may sound simple, but a long path needs to be walked until the test experiment can be performed. The first step is the abstraction of physical reality by means of *formal models*. First, the *digital circuit*, a device that processes input vectors over $\{0, 1\}$ and produces output vectors over the same set, is modelled at a certain level of abstraction. In this thesis, *combinational* circuits are modelled at the *gate level*, and thus regarded as directed acyclic graphs, where the nodes can be either input pins, output pins or logic gates, and the edges are the connections between these components. Each *logic gate* has a specific functionality described by a primitive Boolean function. Thus, the functionality of the whole circuit corresponds to a well-defined Boolean function that maps the circuit's input vectors to the circuit's responses.

Circuits can also have memory elements. Such circuits are called *sequential* and can be modelled as finite-state machines. In many cases, however, it is convenient to ignore the memory elements and to only consider the combinational core. This representation also allows to model the circuit's function over several clock cycles. In this case, several copies of the circuit's combinational core are connected in series, and the *sequential expansion* of the circuit is regarded as one large combinational circuit. In this context, a copy of the circuit at a certain point in time is called a *time frame*.

Also erroneous behaviour needs to be modelled at a certain level of abstraction. This is necessary because the range of possible physical defects is infinite and non-discrete. Therefore, instead of real defects, formal models of defective behaviour are considered during the preparation of the test experiment. Each model of defective behaviour is called a *fault model*. It comprises a set of assumptions that specify the amount of *faults* that need to be considered and the effect that their occurrence induces in a circuit. The most important property of fault models is that they reduce the complexity of the problem. For instance, particle-induced defects cannot be listed exhaustively, as there are infinitely-many possible particle shapes and the exact location of the particle is a continuous parameter [179]. In contrast, fault models define a finite or at least countable number of faults.

The most-used fault model is the *(single) stuck-at fault model* [72, 90], which assumes that a circuit's faulty behaviour stems from exactly one line being either *stuck-at 0* or *stuck-at 1*, i.e. the line permanently has the logic value 0 or 1, respectively, independently of the value driving the line. The stuck-at fault model is the dominant fault model used in practical applications because test patterns generated for stuck-at faults usually cover many permanent defects, and because the model has the advantage that it defines a relatively small number of faults. However, it has been shown that the stuck-at fault model does not accurately reflect several defect types encountered in the currently dominant CMOS technology [91, 110, 161, 11]. For example, shorts and opens account for a large portion of physical defects in CMOS ICs [91, 49], but the stuck-at fault model provides only a very rough approximation to the behaviour caused by these defects, especially considering that a substantial fraction of shorts and opens are resistive [200]. As a consequence, sophisticated *non-standard fault models* have been introduced, especially in order to allow modelling of very complex effects involving multiple lines, like *capacitive crosstalk* [149, 42, 261, 143], *ground bounce* [236] or *power supply noise* [228, 229]. Regarding complex fault models, there are two main approaches. The first consists in modelling specific situations individually. However, this approach usually requires the implementation of dedicated algorithms for each individual model, e.g. [77, 182, 118, 92]. Aside from the cost of implementing different algorithms, also the integration of different methods is complicated because each algorithm may need to model the problem at its own abstraction level. For this reason, also a trend towards *generic* fault models has emerged [64, 144, 119, 158].

Once the models used to represent the circuit and erroneous behaviour have been fixed, *automatic test pattern generation* (ATPG) can take place. Due to the large number of test patterns that would need to be applied if all possible input combinations were considered (2^n for a combinational circuit with n input pins), the dedicated generation of test patterns to cover the set of modelled faults is the most important task in testing.

For a fixed fault model, a test pattern p is said to *detect* a fault f if the response of the fault-free circuit differs from the response of the circuit with the fault f when p is applied to the circuit's inputs. A fault is called *detectable* if a test pattern exists that detects it. If no such pattern exists, the fault is called *undetectable*. ATPG is the process of calculating a test pattern that detects a fault f if f is detectable. An ATPG algorithm is called *complete* if it is guaranteed to find a test pattern if one exists. If a complete algorithm does not find a pattern that detects a fault f, that proves f's undetectability.

Although the fault detection problem for combinational circuits is NP-complete [128], the average complexity of most ATPG instances found in practice is only $\mathcal{O}(n^3)$ [256, 189]. However, ATPG still remains one of the most challenging tasks in testing. In order to overcome the problem's complexity, test pattern generation is usually combined with *fault simulation*. After the generation of a certain number of test patterns, these are simulated in order to determine which not yet targeted faults are also detected by them. Faults detected by simulation can be removed from the target fault list, thus reducing the number of test generation instances to be solved. This technique is known as *fault dropping*. Another positive effect of fault dropping is the reduction of pattern count.

Since the cost of test application depends strongly on the number of test patterns to be applied, *test compaction* techniques are employed to further reduce the number of generated tests without loss of fault coverage. Test compaction can be either *dynamic*, in which case the test search is guided such that each generated test is suitable for the detection of a higher number of faults, or *static*, in which case the size of the generated test set is reduced after the test generation process has been completed.

ATPG algorithms that perform the search based on the circuit's structure are called *structural*. The first steps in structural testing of logic circuits were made by Eldred in 1959 [72], but it was Roth's work at IBM which resulted in the first systematic and complete ATPG method for stuck-at faults — the *D-Algorithm* [201, 202]. The algorithm uses a five-valued logic known as *Roth's logic*. This logic comprises the following values: the logic values 0 and 1, the error values D (assigned to lines that should have the value 1 but have the value 0 due to the presence of the fault) and D′ (assigned to lines that should have the value 0 but have the value 1), and the *unspecified* value x, which is used to represent lines that have not yet been assigned a value by the algorithm. The algorithm assigns the value D or D′ to the fault site depending on whether the target fault is a stuck-at-0 or a stuck-at-1 fault, and computes the values that that assignment implies on other lines. When no further implications can be derived, this branch-and-bound algorithm makes decisions, i.e. it assigns values to yet unspecified lines such that the fault effect is *propagated* towards a primary output and such that the values that have been assigned to lines driven by yet unspecified values can be *justified*. If a decision leads to a conflict, the algorithm *backtracks*, i.e. it corrects wrong decisions and computes the implications of the correction. The algorithm terminates when a fault effect becomes visible at a primary output and all assignments are justified, in which case the fault is detectable, or when the complete search space has been exhausted without finding a solution, in which case the fault is proved to be undetectable.

Two important structural algorithms that can be seen as derivatives of the D-Algorithm are *PODEM* (Path-Oriented Decision Making) [98] and *FAN* (Fan-out Oriented Test Generation) [89, 87], which speed up the process by restricting the locations at which decisions can be made, thus reducing the number of backtracking operations. Some structural algorithms implemented in commercial and proprietary ATPG tools are known to be based on FAN, which is an efficient algorithm able to solve a large number of *easy-to-solve* ATPG instances very fast. Most proposed enhancements of these basic ATPG algorithms [140, 218, 93, 159, 250, 108] are structural as well and rely on learning techniques in order to improve the performance of structural ATPG on *hard-to-solve* ATPG instances.

An alternative to structural ATPG algorithms are *SAT-based* methods, i.e. methods that map the ATPG problem to the problem of *Boolean satisfiability*. This is the problem of deciding whether a Boolean formula is *satisfiable*, i.e. whether its variables can be assigned the values 0 or 1 such that the whole formula evaluates to 1. Software tools used to determine the satisfiability of SAT formulae are called *SAT solvers*. Currently, SAT solvers are used in many fields like planning [136, 96], electronic design automation [166], and verification and test of digital systems [219, 27, 46, 224, 114, 77, 164, 69, 55, 209, 207], especially because many search problems can be converted into SAT problems very efficiently [151].

Given a combinational circuit and a fault f, SAT-based ATPG consists in generating a SAT formula that represents the structure of the circuit both in absence and in presence of the fault. The SAT instance is formulated such that it is satisfiable if and only if f is detectable. If the SAT solver proves that the SAT formula is unsatisfiable, that proves f's undetectability. Conversely, if the SAT solver finds a Boolean assignment that satisfies the SAT formula, f is detectable. Then, the values assigned to the Boolean variables that represent the circuit's primary inputs constitute a test pattern that detects f.

The first approaches to reduce the ATPG problem to a SAT problem were proposed several decades ago [220, 147, 148, 237], but structural algorithms continued to be the standard used in industrial applications due to their better run-times. However, it was shown recently that SAT-based ATPG outperforms structural methods when applied to hard-to-detect and to undetectable faults [242]. The reason for this is that the advances made in SAT solving after the year 2000 were mostly driven by formal-verification problems, i.e. problems in which the equivalence of two models of the same system is to be proved, or in which specific behavioural properties of a system are to be checked. In such problems, the typical workload consists of few, but very hard and usually unsatisfiable SAT instances.

This doctoral thesis covers the contributions to the field of SAT-based test pattern generation made by the thesis's author between 2008 and 2012. Efficient algorithms aimed at enhancing the efficiency of SAT-based ATPG in terms of run-time and test compactness were developed and incorporated into the SAT-based ATPG tool

TIGUAN (**T**hread-parallel **I**ntegrated test pattern **G**enerator **U**tilising satisfiability **AN**alysis), which was implemented from scratch paying special attention to the creation of a particularly efficient and extensible code base. In combination with a pattern-parallel fault-simulator [73], TIGUAN is able to classify all stuck-at faults in three suites of well-known academic benchmark circuits. In particular, TIGUAN classifies all stuck-at faults in a suite of industrial circuits without aborts[1], whereas a commercial, structural tool was not able to classify all faults, even using a high conflict limit. In addition, TIGUAN outperforms the SAT-ATPG tool PASSAT developed at the University of Bremen in regard to run-time, number of aborts and test compactness.

A further important contribution is a new dynamic compaction technique specifically designed for the integration into a SAT-ATPG framework, as the rather high pattern count was traditionally considered to be a major drawback of SAT-based methods. Thanks to the new technique, TIGUAN is able to generate smaller test sets than a commercial, structural ATPG tool for all academic benchmark circuits.

Like the fault simulator, the two SAT solving engines incorporated into TIGUAN were developed within the author's research group, which allowed to implement customisations into the SAT solvers, and to tune their internal parameters so that they could solve the type of SAT instances generated by TIGUAN more efficiently. Moreover:

- The SAT solver *MiraXT* [152, 151] supports thread parallelism. That means that it can distribute the effort of SAT solving among several computation threads that can run in parallel on multi-processor or multi-core systems. The optimal utilisation of this feature was analysed systematically and a *two-stage method* was developed, where faults are processed using different SAT solving parameters depending on the hardness of the produced SAT instances.
- The SAT solver ANTOM [217] supports modern, advanced SAT solving techniques:
    - *Incremental* SAT solving with and without *assumptions* — this means that several SAT instances can be solved using only one instantiation of the SAT solver, and that conflict knowledge learnt during the solving of each SAT instance can be shared with subsequent instances, thus

[1] In order to prevent an excessive grow of the total run-time, it is usual for both structural and SAT-based algorithms to *abort* the processing of single faults when a timeout or a conflict limit has been reached. In that case, those faults remain unclassified. A lower number of aborts stands for a higher algorithm quality.

speeding up the solving process. In addition, initial partial assignments (assumptions) can be passed to the SAT solver. Based on this, a *fault clustering* technique was implemented into TIGUAN, which allowed to further reduce the total time needed to classify all stuck-at faults in a suite of nineteen industrial benchmark circuits by 47.7%. For some circuits a reduction of up to 65.3% was achieved.

- SAT solving with *qualitative preferences* [95, 96, 65] — this is a formal mechanism that allows the user to specify a set of Boolean variables that should be assigned to a preferred value; also the relative importance of those preferences can be laid down. That makes it possible to control with precision the quality of the solutions computed by the SAT solver, and also to formally define solution optimality. This mechanism was employed to implement a SAT-based framework for the generation of test patterns that satisfy user-defined optimisation goals, and the generated test patterns are guaranteed to be optimal. This constitutes a problem class that cannot be solved trivially using structural algorithms.

One of the most important contributions of this thesis is the definition of two generic fault models, the *conditional multiple stuck-at fault model* (CMS@FM) and the *enhanced conditional multiple stuck-at fault model* (ECMS@FM), and the incorporation of their support into the SAT-ATPG tool TIGUAN. As was explained previously, the stuck-at fault model no longer suffices to cover all types of defects that occur increasingly in newer technologies. Using the CMS@FM, it is possible to describe defects that induce faulty behaviour on an arbitrary number of *victim* lines, and to specify the activation conditions for the defect by imposing specific values on a number of *aggressor* lines. A particular feature of CMS@-based SAT-ATPG is its flexibility, which allows the description of ATPG problems with varying degrees of complexity without the need to modify the SAT-ATPG core engine. For example, CMS@-ATPG was used to generate with equal comfort patterns for relatively simple test concepts, like *gate-exhaustive* testing [169, 43], and for realistic defect-based models like *resistive-bridging faults* [198, 199, 197, 75, 78]. In combination with the expansion of sequential circuits, this model can also be used to describe dynamic fault effects.

The ECMS@FM goes even further and supports features not offered by previously existing generic fault models. In combination with SAT solving using qualitative preferences, ECMS@-based SAT-ATPG allows the imposition of *soft constraints* on any number of lines, and thus to control the quality of the generated test patterns with regard to a wide variety of needs. For instance, a set of lines can be chosen and the number of 0 or 1-assignments made to those lines can be maximised or minimised. Some of the example applications based on this principle and discussed in detail in this thesis include:

- The generation of test patterns that maximise the number of primary outputs towards which the fault effect is propagated. Such test patterns have been shown to increase the coverage of transition delay faults [247].
- The generation of test patterns that minimise the number of fault-affected primary outputs, which finds application e.g. in diagnosis [124].
- The generation of test patterns that control precisely the switching activity of a number of selected lines, or globally. For instance, slow-down-crosstalk testing [30] requires that a number of aggressor lines switch in the opposite direction in which the victim line switches, such as to increase the fault effect.

The ECMS@-based SAT-ATPG framework was submitted to a hard test by employing it to generate test sequences for *power droop* testing [245, 177]. Triggered by two different mechanisms over a large number of clock cycles, power droop is a signal integrity issue that leads to localised delay effects. ATPG for power droop constitutes an extremely hard variation of sequential test generation, given that a large number of times frames need to be modelled and that three different optimisation objectives need to be satisfied simultaneously.

Finally, large parts of Tiguan's implementation were optimised and documented such as to provide a C++ library that allowed other researchers to use Tiguan as a SAT-ATPG back-end for various applications.

Organisation of this thesis

Chapters 2 and 3 provide the reader with an introduction to all basic concepts behind the work covered in the thesis. The contents constitute pre-existing knowledge originated in the work of other authors, and references to the original works have been included where appropriate. Chapter 2 concentrates on the basic principles of testing and, in particular, of test pattern generation. Chapter 3 makes a formal introduction of the SAT problem, and discusses the basic algorithms for the solution of this problem. Special attention is given to techniques used by modern SAT solving tools, as the knowledge of these techniques is fundamental to analyse the experimental results presented in later chapters. Then, this chapter focuses on the application of SAT solving to test pattern generation. It introduces the basic principle and reviews previously existing works on SAT-based ATPG.

Chapters 4–6 introduce the SAT-based test pattern generator Tiguan and discuss the techniques that were developed in order to increase Tiguan's run-time efficiency and compaction ability. Chapter 4 begins with a summary of Tiguan's development history. After the formal introduction of the CMS@FM, the chapter resumes with an accurate description of Tiguan's main algorithms, which operate internally on the CMS@ model. Chapter 5 focuses on techniques to improve Tiguan's run-time efficiency. The first part of the chapter gives insight into the algorithms used by the SAT engine MiraXT, and discusses the analysis that was performed in order to evaluate Tiguan's performance on multi-core systems. The second part of the chapter introduces the SAT engine antom and explains the most important differences between antom and MiraXT from the point of view of a SAT-ATPG application. Then, a fault clustering technique that utilises antom's incremental SAT solving is presented and evaluated. Finally, Chapter 6 discusses the dynamic compaction method that was developed for dedicated incorporation into Tiguan. The chapter also analyses the impact that fault list pre-sorting and a conflict limit have on the performance of the dynamic compaction algorithm.

Chapters 7 and 8 address the application of SAT-based ATPG to complex fault models. Chapter 7 gives a thorough motivation for the need of complex fault models, discusses applications of the CMS@FM, and introduces the ECMS@FM, which enables the specification of optimisation goals. The implementation of ECMS@-based SAT-ATPG is explained in detail, and two important applications of the new ECMS@FM are discussed and evaluated. Chapter 8 discusses the previously mentioned application of ECMS@-ATPG to power droop testing, and focuses on strategies to map the original problem to ECMS@-ATPG such as to achieve the best combination of test quality and run-time efficiency.

To conclude the thesis, Chapter 9 discusses further application possibilities for SAT-based ATPG. A C++ library was developed in order to allow client applications to incorporate Tiguan's functionality in the form of a SAT-ATPG back-end engine. The chapter discusses the principles of the interface design, which attempts to achieve maximum flexibility for the client application and efficient communication between the client application and the SAT-ATPG back-end. Then, short summaries of selected works by other authors are presented. These works employ Tiguan as ATPG engine and have relevance in the research areas of process variations and fault tolerance.

Finally, Chapter 10 closes the thesis with a brief summary of the presented topics and a discussion of the role that SAT-based ATPG is expected to play in industrial applications.

Appendix A provides details on the used benchmark circuits.

Own publications

A complete list of all publications by the author of this thesis is provided on pages 223–226. References in the form of a capital letter followed by a number, both enclosed in brackets (for example, [J2]), refer to this publication list.

Note that parts of the work covered in Chapters 4–9 have been previously published in conference or workshop proceedings, as well as in scientific journals. A footnote on the first page of each of these chapters informs the reader which of the author's publications share contents with that specific chapter.

The author's main works, which served as basis for this doctoral thesis, are the following:

- [C16]: This is the seminal work in which the SAT-based test pattern generator Tiguan, the CMS@ fault model and various applications of this fault model were introduced. These topics are covered in Chapter 4 and in Section 7.2.
- [J2]: This is the journal version of [C16].
- [W7]: In this work, the performance of Tiguan based on the utilisation of thread-parallel SAT solving on multi-core architectures was evaluated. This topic is covered in Section 5.1.
- [C13]: This work presented a dynamic compaction method for SAT-based ATPG. This topic is covered in Chapter 6.
- [C7]: In this work, newest SAT solving techniques were incorporated into Tiguan. This allowed the development of a fault clustering technique for the enhancement of Tiguan's run-time performance (this topic is covered in Section 5.2); and the introduction of ECMS@-based SAT-ATPG for the solution of complex test generation problems with optimisation constraints (this topic is covered in Chapter 7).
- [C5]: In this work, the capabilities of the new ECMS@-based ATPG framework were explored by means of the application to test generation for power droop testing. This topic is covered in Chapter 8.
- [C18]: This work (and its journal version [J3]) was originally performed for the author's undergraduate studies (Studienarbeit) and published prior to the author's time as a doctoral student. It provided the necessary background knowledge for the work presented in [C5].

All other publication references (a number enclosed in brackets, e.g. [55]) included in the text refer to the general bibliography list, to be found from page 227 onwards.

2

Introduction to the test of digital circuits

This chapter provides a preliminary introduction to the area of research covered in this thesis. It is not intended to be an exhaustive introduction to the test of digital circuits, but rather to provide the reader with the background knowledge required to understand this thesis, and to establish the terminology used to refer to certain concepts for which different authors might use diverse terms. Further information on the topics covered in this chapter can be found in well-known text books, for example in [12, 34, 129].

2.1 The Boolean algebra

The *Boolean algebra* is an algebra over the set $\mathbb{B} := \{0, 1\}$. In this algebra, one unary and two binary operations are defined:

- the *negation* $\neg$, where $\neg 0 = 1$ and $\neg 1 = 0$,
- the *conjunction* $\cdot$, where $0 \cdot 0 = 0$, $0 \cdot 1 = 0$, $1 \cdot 0 = 0$ and $1 \cdot 1 = 1$,
- and the *disjunction* $+$, where $0 + 0 = 0$, $0 + 1 = 1$, $1 + 0 = 1$ and $1 + 1 = 1$.

Along with these operations, the Boolean algebra is defined by the following axioms:

- *commutativity* — $a + b = b + a$ and $a \cdot b = b \cdot a$ for all $a, b \in \mathbb{B}$,
- *associativity* — $a + (b + c) = (a + b) + c$ and $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ for all $a, b, c \in \mathbb{B}$,
- and *distributivity* — $a + (b \cdot c) = (a + b) \cdot (a + c)$ and $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ for all $a, b, c \in \mathbb{B}$.

By combination of the three basic operations, more operations can be defined. The most relevant combination is the *exclusive disjunction* $\oplus$, a binary operation defined by $a \oplus b = (a \cdot \neg b) + (b \cdot \neg a)$ for all $a, b \in \mathbb{B}$. Note that commutativity and associativity both hold for the exclusive disjunction. Also, $0 \oplus 0 = 1 \oplus 1 = 0$ and $0 \oplus 1 = 1$.

In addition, several important properties of the Boolean algebra can be derived from the main axioms. Some examples follow:

- *absorption* — $a \cdot (a + b) = a$ and $a + (a \cdot b) = a$ for all $a, b \in \mathbb{B}$,
- *complement rule* — $a + \neg a = 1$ and $a \cdot \neg a = 0$ for all $a \in \mathbb{B}$,
- and *De Morgan's law* — $\neg(a + b) = \neg a \cdot \neg b$ and $\neg(a \cdot b) = \neg a + \neg b$ for all $a, b \in \mathbb{B}$.

2.2 Circuits

2.2.1 Modelling levels

At any level of abstraction, a *digital circuit* can be seen as a device that processes input data and produces output data, where both the input and output data are represented by vectors over $\mathbb{B}$. The circuit's *function* is defined by the lengths of the input and output vectors and by the mapping from the input to the output domain. Hence, the simplest representation of a circuit is the *truth table*. For example, Figure 1 shows the truth table of a half-adder, a circuit that takes two arguments a and b and produces two outputs c and d such that the sequence cd is the binary representation of $a + b$.

inputs		outputs	
a	b	c	d
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Figure 1. Truth table of a half-adder

However, truth tables are of little practical use for large circuits and for tasks that depend not only on the circuit's function but also on its implementation. Hence, numerous other ways of modelling a circuit are considered in the literature and in practice. In order to organise the different types of models, different *levels of* *abstraction* (also called *modelling levels*) are distinguished, but different authors may define different numbers of modelling levels depending on what they want to illustrate [179]. For instance, Hayes defines in [111] three *design levels*: processor level, register level and gate level.

In [253], where the focus lies on design verification and test, a hierarchy composed of four design levels is given. The design process is viewed as a series of transformations that map design descriptions from higher levels to lower levels. The highest modelling level is the *behavioural* or *architecture level* which focuses on the functionality of the modelled circuit or system. Given a design specification, the behaviour of the system is specified using algorithmic notation, for example in the form of a *hardware description language.* The next lower level is the *register-transfer level*, which contains more structural information in terms of the implemented logic functions, data and control paths. The implementation of logic functions is modelled in the next lower level, the *logical* or *gate level.* In this level, a circuit is composed of a number of *logic gates*, where each logic gate is a component with a specific functionality that is defined by a Boolean function. For instance, an AND gate is a component with two inputs a and b and an output c, where $c = a \cdot b$. Gates are connected to each other by *signal lines*. By traversing the circuit in topological order, it is possible to construct a Boolean function that represents the whole circuit. Finally, the lowest level is the *physical* or *transistor level.* In the same way in which circuits can be modelled by gates and connections between gates, gates are internally modelled by transistors and by connections between different transistors or between transistors and power sources (V_{DD}) or ground. The transistor level can be seen as a refinement of the gate level, and sometimes a mixture of both levels can be used for testing. For instance, faults can be modelled at transistor level in order to reflect defects that are more realistic from a physical point of view, but the rest of the circuit may be modelled at gate level such as to avoid unnecessary overhead, for example during simulation tasks.

Some authors also consider a lower level, the *layout level* [129]. In this level, the circuit description encompasses line widths, inter-line and inter-component distances, as well as device geometries.

Throughout this thesis, gate-level modelling is assumed. The advantage of this level is that it can be regarded as technology-independent. In most process technologies, synthesis tools have readily available libraries containing mappings for the basic logic gates. Hence, it is relatively easy to transform a gate-level description into a (technology-dependent) transistor-level description.

2.2.2 Gate-level net lists

A *digital combinational* circuit C is a device with n inputs and m outputs, whose behaviour can be uniquely specified by a Boolean function $\varphi_C : \mathbb{B}^n \to \mathbb{B}^m$.

At the gate level, a combinational circuit is represented by a *gate-level net list*, a directed acyclic graph (N, L), where N is the set of *nodes* and L is the set of *edges*. The set of nodes is composed of the following sub-sets:

- G, the set of *logic gates*,
- F, the set of *fan-out nodes*,
- I, the set of *inputs*, and
- O, the set of *outputs*.

The edges represent connections between nodes. They are called *signal lines*, *wires* or *nets*. The number of ingoing and outgoing edges of each node is determined by the node's type. The node sets I and O represent the inputs and outputs of the circuit, respectively. Hence, the former have no ingoing edges and exactly one outgoing edge, while the latter have exactly one ingoing edge and no outgoing edges.

Logic gates have exactly one output (outgoing edge) and one or more inputs (ingoing edges). In this thesis, the output of a gate together with the set of all its inputs are referred to as that gate's *ports*.

Each logic gate $g \in G$ implements a Boolean function $\varphi_g : \mathbb{B}^k \to \mathbb{B}$, where k is the gate's number of inputs. Which function is implemented by g is specified by g's *type*. For instance, a two-input AND gate implements the logic conjunction, i.e. for every pair of inputs $a, b \in \mathbb{B}$, the output produced by the gate equals $a \cdot b$.

Figure 2 lists all basic gate types considered in this thesis, along with the Boolean function they implement and their symbol in graphical representations of circuits. Although these gates suffice to construct circuits that implement any Boolean function, versions of AND, NAND, OR and NOR gates with more than two inputs are also considered by many authors. However, such gates do not require special attention in the description of algorithms, because their functionality can be expressed in generalised form independently of the number of gate inputs. Thus, the functionality of buffers and inverters can be expressed based on one single parameter called *inversion*. Given an input v, the gate produces the output $\neg v$ if it is inverting (INV), or the output v if it is not inverting (BUF).

Analogously, the functionality of an AND, NAND, OR or NOR gate g can be described using only two parameters, the gate's *inversion* and the gate's *controlling value* $\text{CV}(g)$.

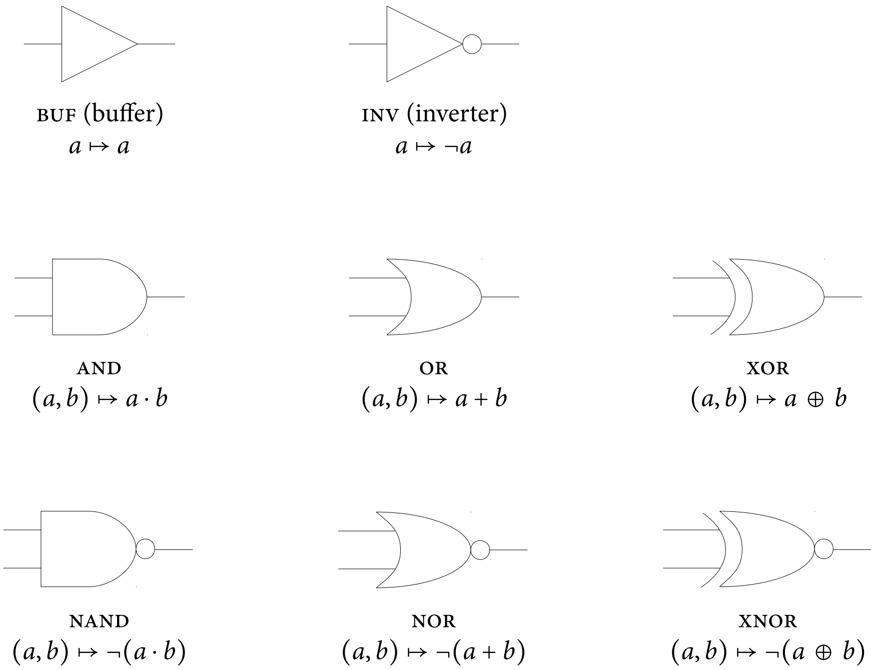

BUF (buffer)
$a \mapsto a$
INV (inverter)
$a \mapsto \neg a$
AND
$(a, b) \mapsto a \cdot b$
OR
$(a, b) \mapsto a + b$
XOR
$(a, b) \mapsto a \oplus b$
NAND
$(a, b) \mapsto \neg(a \cdot b)$
NOR
$(a, b) \mapsto \neg(a + b)$
XNOR
$(a, b) \mapsto \neg(a \oplus b)$

Figure 2. Gate types

The Boolean inverse of CV(g) is called g's *non-controlling value* (NCV(g)). If at least one input of g has the logic value CV(g), then g produces the output value CV(g) independently of the logic value on all other inputs (or the output NCV(g) if g is inverting). g can only produce the output NCV(g) (CV(g) if g is inverting) if all its inputs have the logic value NCV(g). Table 1 lists the inversion, and the controlling and non-controlling values of these four types of gates.

Table 1

Gate parameters

gate type	inverting	controlling value	non-controlling value
AND	no	0	1
NAND	yes	0	1
OR	no	1	0
NOR	yes	1	0

The functionality of XOR and XNOR gates cannot be expressed in this form. Hence, while gate-level-net-list-based algorithms can process all other gate types using a generic procedure that works only in function of the gate's inversion and controlling value, XOR and XNOR gates need to be processed separately.

Since $a \oplus b = (a \cdot \neg b) + (b \cdot \neg a)$ and $a \oplus b = \neg(\neg(a \cdot \neg(a \cdot b)) \cdot \neg(b \cdot \neg(a \cdot b)))$ for all $a, b \in \mathbb{B}$, XOR and XNOR gates can also be replaced by equivalent sub-circuits composed of two inverters, two AND gates and an OR gate, or by equivalent sub-circuits composed of four NAND gates, without changing the logic functionality of the circuit. However, this replacement can alter the timing of the circuit and should thus be used only for algorithms that operate only on the logic functionality of the circuit.

Going back to the definition of the graph (N, L), the set of fan-out nodes F is composed of nodes with one ingoing edge and at least two outgoing edges. The ingoing edge and all outgoing edges of a fan-out node are regarded as one signal that is branched to distribute the output of one gate to multiple other gates. The ingoing edge is called the fan-out's *stem*, while the outgoing edges are called the fan-out's *branches*.

Figure 3 shows an example circuit and illustrates the naming conventions observed in this work. Inputs and gates are denoted by lower case letters. For simplicity, lines are given their own identifiers only when strictly necessary. When that is not the case, they are referred to using the identifier of the gate at which they originate. When necessary, the branches of a fan-out node are denoted by the stem's identifier, but with an index. Circuit outputs are denoted by the identifier of their ingoing edge. The shown circuit represents the functionality of a half-adder (see also Figure 1). Gate c implements the Boolean function $(a, b) \mapsto a \cdot b$ and gate d implements the Boolean function $(a, b) \mapsto a \oplus b$. Hence, the whole circuit implements the Boolean function $\mathbb{B}^2 \to \mathbb{B}^2$, $(a, b) \mapsto (a \cdot b, a \oplus b)$.

Let a line connect the output of a gate g_1 to one of the inputs of a gate g_2. Then, g_2 is called a *successor gate* of g_1, and g_1 is called a *predecessor gate* of g_2.

If the output of g_1 is connected to a fan-out node, then all gates connected to the fan-out branches of that node are g_1's successors, and g_1 is a predecessor of all those gates.

The sequence of gates $g_1, \ldots, g_k$ is called a *path* from g_1 to g_k if g_{i+1} is a successor gate of g_i for all $i = 2, \ldots, k$. A path is said to be *complete* if it starts at a circuit input and ends at a circuit output. A path that is not complete is called *partial*. For a gate g_i, the input of g_i that is connected to g_{i-1} is called the *on-path* input of g_i, as that line belongs to the path. All other inputs of g_i are called *off-path* inputs.

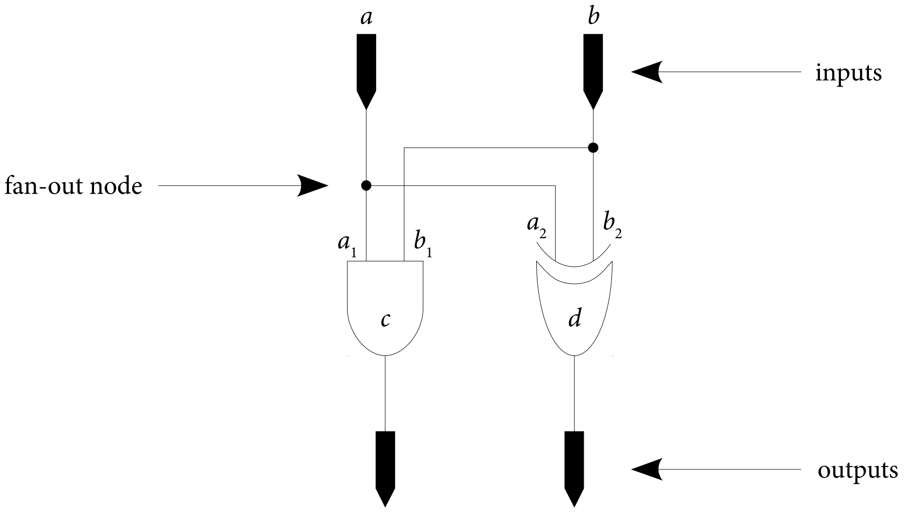


Figure 3. Gate-level half-adder

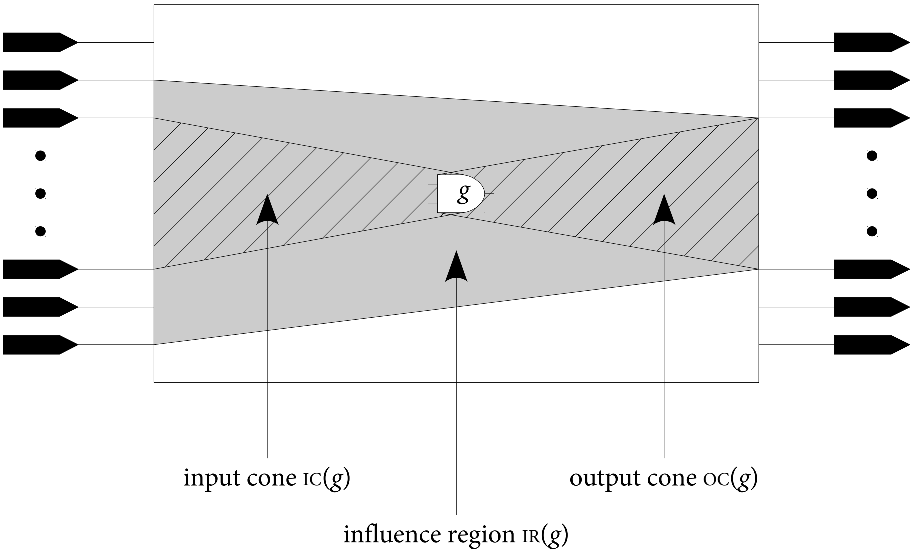


Figure 4. Cones of influence

The *output cone* of a gate g ($\textsc{oc}(g)$) is defined as the set of all gates that belong to a path between g and any circuit output, while the *input cone* of g ($\textsc{ic}(g)$) is defined as the set of all gates that belong to a path between any circuit input and g. Let $g_1, \dots, g_n$ be all circuit outputs contained in $\textsc{oc}(g)$. Then, the set $\textsc{ir}(g) := \textsc{ic}(g_1) \bigcup \cdots \bigcup \textsc{ic}(g_n)$ is called g's *influence region* (Figure 4).

The number of fan-out nodes and the average number of fan-out branches per node are a factor that strongly influences the run-time efficiency of many algorithms that work on gate-level net lists. Hence, some algorithms partition the circuit into *fan-out-free regions* (FFR), i.e. sub-circuits without fan-out nodes, and then process each FFR separately. Each FFR has the form of a tree, where its *root gate* is connected either to a circuit output or to a fan-out node. The partition of a circuit into FFRs is unique.

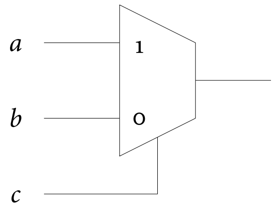
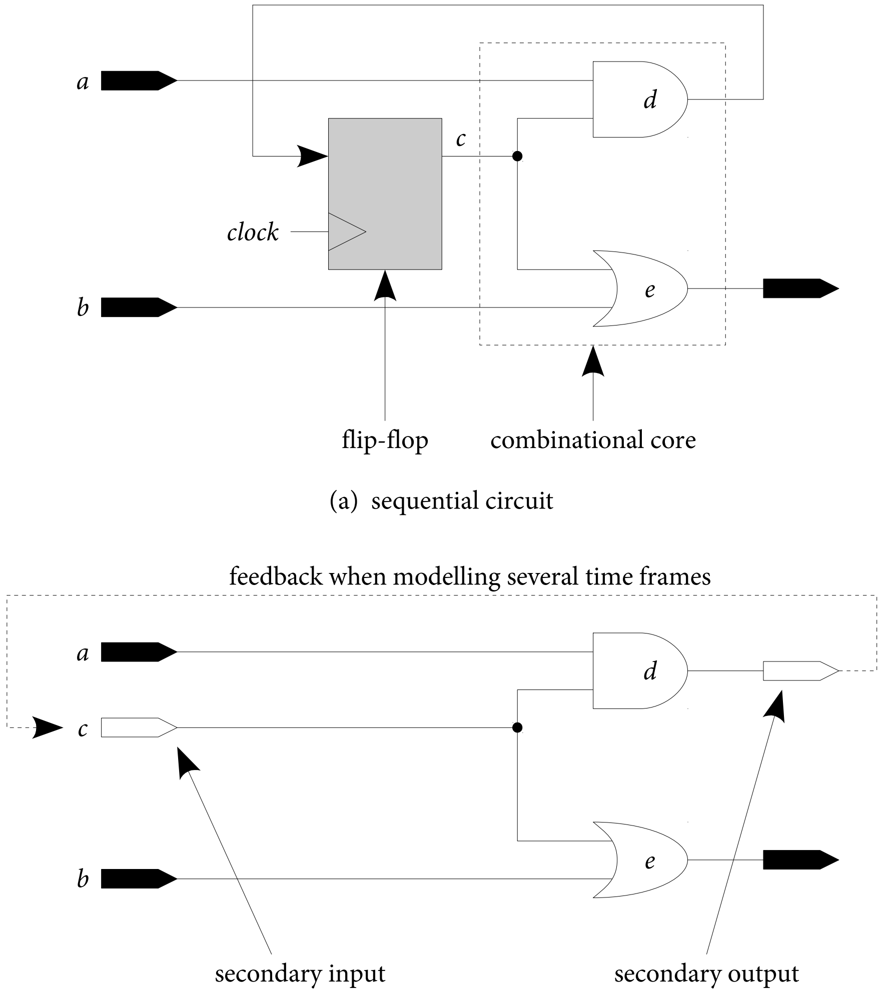


FIGURE 5. A TWO-INPUT MULTIPLEXER

To close this section, Figure 5 shows the symbol used in this thesis to represent *multiplexers*. A two-input multiplexer is a circuit that implements the Boolean function $(a, b, c) \mapsto (c \cdot a) + (\neg c \cdot b)$, i.e. the control input c selects which of the two inputs a and b is passed to the output.

2.2.3 SEQUENTIAL CIRCUITS

A *digital sequential* circuit is a circuit that contains cycles due to the presence of memory elements, mostly *flip-flops*. Flip-flops are *clocked*, i.e. they are connected to a device that generates a *clock signal*. The clock signal oscillates between logic 1 and logic 0, normally with a 50% duty cycle, and is used to synchronise all flip-flops. These are designed such that they can store a new value only while the clock is high (or only when the clock is low, depending on the implementation). A *clock cycle* is composed of one *falling* (a 1→0 transition) and one *rising edge* (a 0→1 transition) of the clock and its length is called *clock period*. This length is denoted by T_{clk}. In normal operation mode, new input vectors are applied to the sequential circuit once per clock cycle.



(a) sequential circuit

(b) simplified representation

Figure 6. Example sequential circuit

In contrast to combinational circuits, the functionality of sequential circuits cannot be simply specified by a Boolean function. Instead, a sequential circuit C with n inputs, m outputs and k flip-flops can be regarded as an implementation of a finite-state machine [171] with 2^k or less states. The states are encoded by the data stored in the flip-flops, while the *combinational core* of the circuit computes the output function $\varphi_C : \mathbb{B}^n \times \mathbb{B}^k \to \mathbb{B}^m$ and the transition function $\tau_C : \mathbb{B}^n \times \mathbb{B}^k \to \mathbb{B}^k$, which depend both on the inputs and on the present state. The φ_C-values correspond to the circuit's outputs, while the τ_C-values are stored back into the flip-flops.

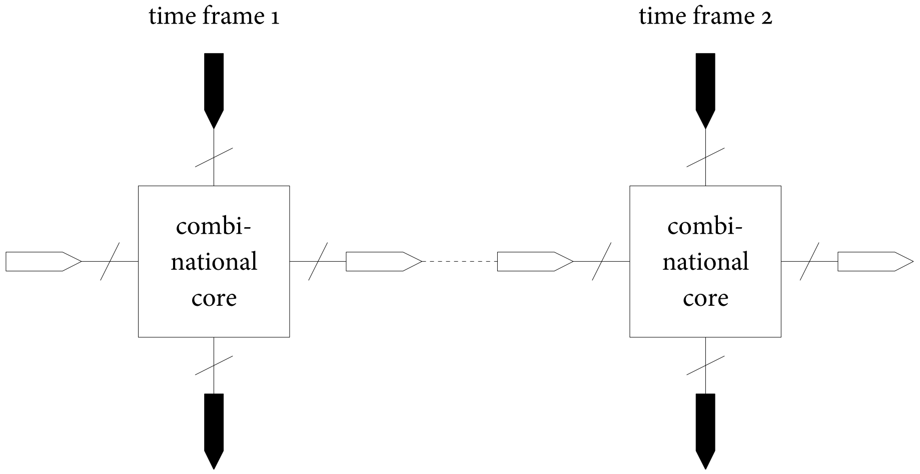


Figure 7. Sequential expansion

An example sequential circuit is shown in Figure 6 (a). In this example, $n = 2$, $m = 1$ and $k = 1$. The corresponding state machine has two states, 0 and 1, encoded by c. φ_C maps (a, b, c) to $b + c$ and τ_C maps (a, b, c) to $(a \cdot c)$.

When fault simulation or test pattern generation are applied to sequential circuits, it is often convenient to ignore the flip-flops and to only consider the combinational core (Figure 6 (b)). The outputs of flip-flops (c in the example) are then treated as additional inputs of the combinational core. These are called *pseudo-primary* or *secondary inputs* (SI). The inputs of memory elements (d in the example) are treated as additional outputs of the combinational core. These are called *pseudo-primary* or *secondary outputs* (SO). Regular inputs and outputs (a, b and e) are then called *primary inputs* (PI) and *primary outputs* (PO), respectively. Instead of the output function φ_C and the transition function τ_C, only one global Boolean function is considered: $\varphi_C^{\text{seq}} : \mathbb{B}^{n+k} \to \mathbb{B}^{m+k}$, which is computed by the combinational core. In the example, φ_C^{seq} maps (a, b, c) to $(b + c, a \cdot c)$.

This representation also allows to model the circuit's function over several clock cycles. In this case, several copies of the circuit's simplified representation are connected in series, where the correspondence between secondary outputs and secondary inputs that are connected to the same flip-flop in the original circuit has to be observed (Figure 7). In this context, a copy of the circuit at a certain point in time is called a *time frame*.

2.3 Fault models

2.3.1 Defects, faults and errors

In engineering, models bridge the gap between physical reality and mathematical abstraction. This is especially true in the test of digital circuits since the range of possible physical defects is infinite and non-discrete. For this reason, the modelling of faulty behaviour is one of the most important issues that need to be considered for the development and application of test algorithms.

Bushnell and Agrawal distinguish between three different terms: defects, faults and errors [34]. A *defect* is the unintended difference between the implemented hardware and its intended design. However, in this case, the term only refers to defects that result from the imperfection of the manufacturing process, not to design defects. Some typical defects in VLSI chips are [122]:

- process defects — missing or broken contacts, parasitic transistors, shorts, oxide breakdown, etc.,
- material defects — cracks, crystal imperfections, surface impurities, etc.,
- age defects — dielectric breakdown, electromigration, etc.,
- package defects — contact degradation, seal leaks, etc.

A *fault* is a formal representation of the defect, while the wrong response of a defective system is an *error*. For example, assume that one of the inputs of an AND gate is shorted to ground. The unintended short is the defect. It can be represented by a *stuck-at-0 fault*, i.e. a formal model that assumes that the shorted line has always the logic value 0 independently of the value produced by the gate driving the line. An error occurs if the value 1 is applied to both inputs of the gate. Then, the gate produces the erroneous output value 0 instead of the expected 1.

A *fault model* is a set of assumptions that specify the amount of faults that need to be considered and the effect that their occurrence induces in a circuit. Not all fault models try to reflect physical reality with accuracy. In fact, the main aim of fault models is to reduce the complexity of the problem. For instance, particle-induced defects cannot be listed exhaustively, as there are infinitely-many possible particle shapes and the exact location of the particle is a continuous parameter [179]. For this reason, a model that attempts to represent every possible particle-induced defect is not feasible. Instead, fault models define a finite or at least countable number of faults, where the behaviour of each fault corresponds roughly to the behaviour of a set or a class of realistic defects.

The following list names some of the possible effects of manufacturing defects [161, 109, 179]:

- The Boolean function φ_C computed by C can be altered.
- The function computed by the circuit may become non-Boolean, i.e. some output of the circuit produces a voltage that cannot be clearly interpreted as logic 0 or logic 1.
- Some lines in the circuit can show a memory behaviour, thus making a combinational circuit sequential.
- The timing of the circuit can be affected.

2.3.2 The stuck-at fault model

The most-used fault model is the *(single) stuck-at fault model* (SAFM) [72, 90]. This model assumes that a circuit's faulty behaviour stems from exactly one line being either *stuck-at 0* (s-a-0) or *stuck-at 1* (s-a-1), i.e. the line permanently has the logic value 0 or 1, respectively, independently of the value produced by its driving gate or the value on its source fan-out stem. Hence, the number of possible faults is linear in the number of lines. Nevertheless, empirical experience has showed that a test pattern set generated for the SAFM can achieve a high coverage of permanent defects. A related fault model is the *multiple stuck-at fault model*, which allows several lines to be simultaneously stuck at a certain value. However, test sets generated for this fault model rarely achieve a considerably better coverage, while test pattern generation is more complicated due to the larger number of faults.

2.3.3 Delay fault modelling

As explained at the end of Section 2.3.1, some defects do not modify the logical behaviour of the circuit. Instead, they affect the timing of the circuit. Such defects cannot be covered using the SAFM or other *static* fault models. In this section, the most important *delay fault* models are introduced. These are the *gate delay* fault model (GDFM) [37, 188], the *path delay* fault model (PDFM) [231, 155] and the *segment delay* fault model (SDFM) [113].

The GDFM assumes that a single gate is affected, and that the gate propagates either *rising* (0→1) or *falling* (1→0) transitions too slowly. The advantage of this model is the very small number of faults it defines. However, the assumption that only one site is affected while the rest of the circuit remains unaffected may not always be realistic, as delay faults often arise from variations in the manufacturing process, and such variations tend to affect the whole circuit.

Under the PDFM, a complete path is assumed to be faulty. Here, a path is defined as free of timing defects if, for every pair of test patterns that induces a falling (or a rising) transition at the beginning of the path, the correct logic value stabilises at the end of the path in less time than the duration of a clock cycle. The *slack* of a path is the difference between the longest possible delay of the fault-free path and the clock period.

The PDFM reflects reality better than the GDFM, as it models the accumulated effect of delay variations along a path. However, in the worst case, the number of paths in a circuit is exponential in the number of fan-out nodes. Hence, the generation of test patterns for every path is impractical. The solution to this problem consists in identifying a certain number of most critical paths, i.e. paths with a small slack, and generating test patterns only for those paths [190, 208, 209, 130, 211]. As an alternative solution, the SDFM has been proposed, according to which only partial paths are considered. A comparative study on delay fault models was presented for example in [160].

2.4 Test application and fault coverage

2.4.1 Definitions

Let C be a combinational circuit with n inputs and m outputs, and let φ_C be the Boolean function implemented by C. Let f be a fault according to a fault model that represents defects that affect the Boolean function computed by C. Then, the faulty-case Boolean function is denominated by φ_C^f.

Let $p \in \mathbb{B}^n$ be an input vector[2] (input vectors are also called *input patterns*, *test patterns* or *tests*). p is said to *detect* f, if $\varphi_C(p) \neq \varphi_C^f(p)$. That means, if a manufactured circuit instance contains a defect that behaves like fault f, the defect's presence can be detected by applying the test pattern p to the circuit and observing whether the circuit's response is correct. Note, however, that this reasoning cannot be reversed. A wrong response to the application of p does not automatically imply the presence

[2] For better readability, a test pattern $p \in \mathbb{B}^n$ will be written as a sequence $b_1 \cdots b_n$ rather than as a vector $(b_1, \ldots, b_n)$. Each component b_i is called a *bit*.

of f. It could be any fault that behaves in the same way as f under the application of p, but that may behave differently under the application of other input patterns.

This definition can be extended to a set of faults $F := \{f_1, \ldots, f_{r_F}\}$ (usually called a *fault list*) and a set of test patterns $P := \{p_1, \ldots, p_{r_P}\} \subseteq \mathbb{B}^n$ (called a *test set*). A fault $f_i \in F$ is detected by P if P contains at least one test pattern that detects f_i.

Let f be a fault, and let P_{exh} be the *exhaustive test set*, i.e. the set that comprises all 2^n test patterns. If P_{exh} detects f, f is called a *detectable* fault. Faults that are not detectable (i.e. no test pattern that detects them exists) are called *undetectable* or *redundant* faults.

Two faults f_1 and f_2 are called *equivalent* if and only if $\varphi_C^{f_1} = \varphi_C^{f_2}$. That means, if f_1 and f_2 are equivalent, then all patterns that detect f_1 also detect f_2 and vice versa.

The *fault coverage* is a measure to grade the quality of a test set. In its most general form, it is defined by

$$\text{fault coverage of a test set } P = \frac{\text{number of faults } P \text{ detects}}{\text{number of faults defined by fault model}} \cdot 100\%.$$

This section closes with the definition of two important terms that are often used in this thesis and that shall not be confused with each other — *fault activation* and *fault excitation*. A fault is said to be *excited* by a test pattern p if p satisfies the conditions that allow a fault effect to become visible at the fault site. Fault excitation is a necessary condition for fault detection. For instance, a stuck-at-1 fault is excited if p induces the value 0 on the fault site. In contrast, if p induces the value 1 on the fault site, the presence of the fault cannot be detected as the induced fault-free behaviour does not differ from the faulty behaviour.

The term “fault activation” is used in the context of conditional fault models which define that a number of victim lines display erroneous behaviour only if a number of aggressor lines satisfy certain conditions. One such model is the CMS@ fault model which will be introduced in detail in Section 4.2. A fault is said to be *activated* by a test pattern p if p satisfies the conditions on the aggressor lines which are necessary to incite the victim lines to faulty behaviour. Note that a test pattern can activate and yet not excite a fault.

2.4.2 Test application

Figure 8 illustrates the basic test application scheme. The *automatic test equipment* (ATE) has a memory module in which the test set (obtained by test-pattern-generation algorithms, see Section 2.7) and the expected fault-free responses (obtained by simulation, see Section 2.6) are stored prior to the test start. Then, the ATE applies every test pattern to each actual manufactured circuit (*circuit under test* — CUT) and compares the expected fault-free response to the response produced by the CUT. If a difference is observed, the CUT is identified as faulty.

In addition to the test application, also diagnosis can be applied to the CUT. *Diagnosis* is the process of locating the physical defect that caused the observed erroneous behaviour. In a printed circuit board, for instance, chips identified as faulty can be replaced and open lines or shorts between pins can be repaired via resoldering. In contrast, digital VLSI chips are usually unrepairable, but diagnosis can be performed on a sample of faulty chips in order to determine the root causes behind common failures or performance problems. Knowledge about the causes can be used to modify one or more steps of the design or fabrication process such as to increase the production yield or the performance of the fabricated chips. The *logic diagnosis* of failed chips consists in analysing the faulty responses. One type of analysis methods uses *fault dictionaries*, i.e. a mapping between faulty responses and the sets of faults that can cause each of the responses. For more information, see e.g. [129].

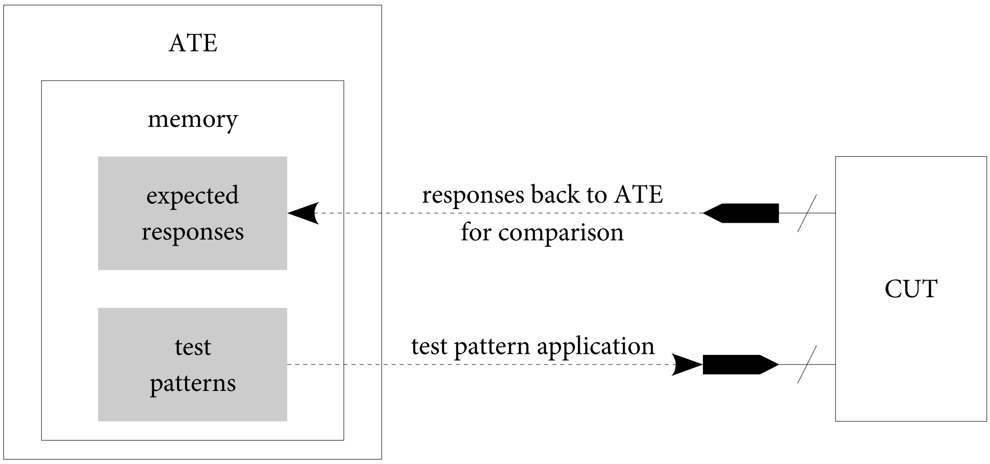


Figure 8. Test application

2.4.3 Two-pattern testing

Testing for delay faults requires *two-pattern testing*, i.e. the application of two successive test patterns (*a test pair*) to the CUT. Given a test pair $\mathfrak{p}$, the first pattern (*initialisation pattern*) is denoted by $p^{(1)}$ and the second pattern (*propagation pattern*) is denoted by $p^{(2)}$. $p^{(1)}$ brings the CUT into a known and stable state. $p^{(2)}$ excites the fault and propagates the fault effect to a circuit output by inducing a rising or falling transition at one or more inputs of the CUT. Assuming that $p^{(2)}$ is applied at time t, the circuit is defect-free if its internal state (flip-flop contents) and its outputs comply with the specification at time $t + T_{\text{clk}}$.

For example, a test pair $\mathfrak{p}$ must meet the following conditions in order to detect a slow-to-rise GDF at a gate g:

- $p^{(1)}$ must induce the logic value 0 on g's output.
- $p^{(2)}$ must induce the logic value 1 on g's output, thus launching a rising transition at the fault site.
- Both $p^{(1)}$ and $p^{(2)}$ must sensitise a path from the fault site to a circuit output, thus making the fault effect observable.

A path is said to be *sensitised* by a test pair $\mathfrak{p}$ if the application of $\mathfrak{p}$ induces a rising or a falling transition on the output of all gates that belong to the path. In order for a path to be sensitised by $\mathfrak{p}$, $\mathfrak{p}$ must induce specific values on the off-path inputs of each multiple-input gate that belongs to the path. Which values are to be induced is determined by the used *sensitisation condition* and influences the quality of the test pair $\mathfrak{p}$ with respect to the probability that it detects the fault if another delay defect is present simultaneously. Several authors have defined alternative sensitisation conditions, which has resulted in a hierarchy that includes *hazard-free robust*, *robust*, *strong non-robust*, *weak non-robust*, and *functional sensitisation* [129, 196, 207]. Hazard-free robust sensitisation requires that all off-path inputs of each gate g that belongs to the path have stable $\textsc{ncv}(g)$-values during the application of $p^{(1)}$ and $p^{(2)}$. This sensitisation condition results in a test of highest quality, as the test is guaranteed to detect the fault independently of other delay defects that may occur simultaneously. However, guaranteeing signal stability on all off-path inputs imposes specific conditions on a large amount of signals, thus reducing the probability that a test pair with such a property exists. The next-weaker sensitisation condition is robust sensitisation which requires that $p^{(1)}$ and $p^{(2)}$ both induce the value $\textsc{ncv}(g)$ on every off-path input of each gate g, but which allows instabilities in form of short temporal signal changes on these lines. The next type of sensitisation, strong non-robust sensitisation, is even weaker, as it only requires that each off-path input eventually stabilises to $\textsc{ncv}(g)$ in order to ensure the propagation of the fault effect under the application of $p^{(2)}$, but no conditions are imposed on $p^{(1)}$ regarding the values induced on off-path inputs. Hence, such a test pair is easier to find, but the test can be invalidated if other delay defects are present simultaneously. Finally, weak non-robust and functional sensitisation even relax the conditions imposed on on-path lines, but these types of sensitisation are not further considered in this thesis.

The practical application of two-pattern tests to combinational circuits represents no additional challenge as compared to single-pattern tests. In contrast, the application of two-pattern tests to sequential circuits is considerably more difficult. In the most general case, the secondary inputs and secondary outputs of a sequential circuit are not externally accessible, which may prevent secondary inputs from adopting *necessary* values, i.e. values that are required to sensitise the path or to induce the proper transition at the fault site. In addition, if the fault effect can only be propagated to a secondary output, the fault effect becomes unobservable.

The standard approach to allow the application of two-pattern testing to sequential circuits is *scan design*[3]. Figure 9 illustrates the principle of scan design. A sequential circuit is given two additional primary inputs *scanin* and *scanenable*, and one additional primary output *scanout*. Additional multiplexers are introduced in order to control the way in which the flip-flops are utilised. When *scanenable* is inactive (i.e. *scanenable* = 0), the multiplexers are switched such that the combinational core's secondary outputs are connected to the flip-flop inputs and the flip-flop outputs to the core's secondary inputs, which makes the circuit operate in normal mode. When *scanenable* is activated, the flip-flops form a shift register called the *scan chain*. Then, arbitrary values can be shifted into the flip-flops via *scanin*, while their content can be shifted out over *scanout*.

When all flip-flops are part of the scan chain (*full scan*), applying arbitrary initialisation patterns is possible. However, since the two patterns of a test pair have to be applied in consecutive clock cycles, it is not possible to apply arbitrary propagation patterns. Several solutions have been proposed in order to address this issue. The most important solutions are enhanced scan [63], skewed-load testing (also called *launch-on-shift*) [214] and broad-side testing (also called *launch-on-capture*) [215].

Enhanced scan uses special flip-flops that allow to shift in arbitrary values for the propagation pattern. In this case, test pattern generation can be easily done by treating the secondary inputs and outputs like primary ones. But the hardware

[3] *Design for test* (DFT) stands for a number of techniques that modify the circuit's design in order to facilitate the test of the manufactured circuit. Scan design is the most important DFT technique.

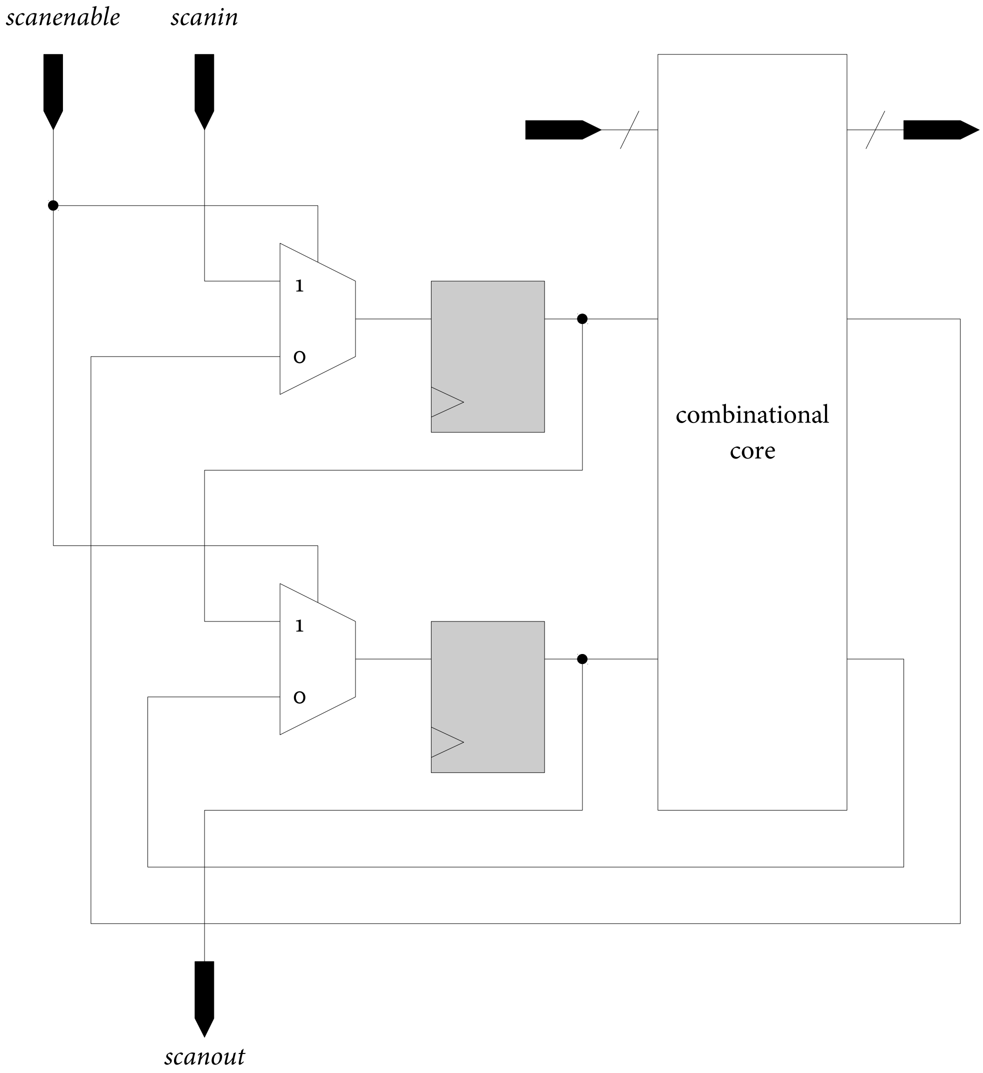


FIGURE 9. SCAN DESIGN

overhead of this technique is very high. In *skewed-load testing*, the propagation pattern is obtained by shifting the initialisation pattern by one position; and in *broad-side testing*, the flip-flop contents after the application of the initialisation pattern serve as the propagation pattern. In these two cases, test generation has to consider the constraints that relate the propagation pattern to either the initialisation pattern or to the circuit's response. This usually makes the test generation problem instances harder to solve.

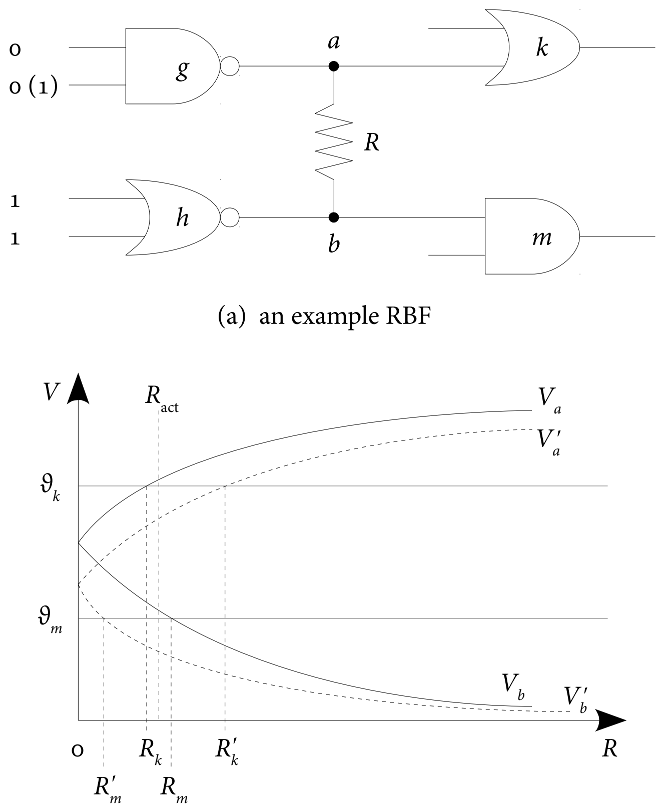
2.5 Resistive fault models

Defects that affect the interconnections of components are usually modelled as opens and shorts. While *opens* correspond to broken lines, *shorts* are formed by connecting lines not intended to be connected. The logical fault that represents a short between internal lines of the circuit is called a *bridging fault*. The two most simple types of bridging fault models are AND *bridges* and OR *bridges*. The former assumes that if two lines l_1 and l_2 driven by logical values v_1 and v_2, respectively, are bridged, then both lines adopt the value $v_1 \cdot v_2$, i.e. the line with a 0-value *dominates* the other line; the latter model assumes that both lines adopt the value $v_1 + v_2$, i.e. the line with a 1-value dominates the other line [12].

However, these two bridging models are too simple to reflect the behaviour of realistic short defects. Hence, more advanced fault models have been proposed [15, 23, 85, 84, 167, 195], but these modelling approaches disregard the fact that a substantial fraction of short defects are resistive [200] and assume a resistance of 0Ω. The reason for this simplification is that the resistance of a bridge is a continuous parameter that is not known in advance and cannot be predicted as it depends on highly variable characteristics of the particle causing the short, like its size, shape, conductivity and exact location. Furthermore, the actual resistance of a resistive bridge influences the behaviour of the defect. For instance, a defect that can be detected by a given test pattern may remain undetected by the same pattern in a different circuit instance in which the resistance is different. Hence, in order to perform realistic test generation and fault simulation for *resistive-bridging faults* (RBF), the concepts of detectability, undetectability and fault coverage need to be adapted when non-zero resistances are modelled [179].

A solution to this problem was presented by Renovell et al. [198, 199, 197], who introduced the concept of *analogue detectability intervals* (ADI). For each RBF f and each test pattern p, an ADI $[R_1, R_2]$ is defined such that p is guaranteed to detect f if and only if $R_1 \leq R \leq R_2$, where R is the actual resistance of the bridge. Hence, the detectability of f is defined individually for each test pattern that detects f for a given resistance interval, and the overall *detection probability* of f can be computed taking into account the probability distribution for the resistance R and f's detectability for each of its ADIs.

The analysis that is performed in order to determine a given RBF's ADIs is explained here by means of an example. Figure 10 (a) shows an example RBF that bridges two nodes a and b, which are the output of the NAND gate g and the output of the NOR gate h, respectively. The fault is excited by imposing opposite logic values on a and b, for instance by applying the value 0 to both inputs of gate g, and the value 1 to



(a) an example RBF

(b) corresponding detectability intervals

Figure 10. Analogue detectability intervals of resistive bridging faults

both inputs of gate h. In absence of the bridge, these input values lead to logic 1 on a and logic 0 on b.

In presence of the bridge, the voltages V_a and V_b measured on nodes a and b, respectively, depend on the bridge's resistance R. For $R = 0\Omega$, there is some intermediate identical voltage on both lines. For $R = \infty$, V_a equals V_{DD} and V_b equals 0V, as if the bridge were not present. The solid curves in Figure 10 (b) depict a possible distribution of V_a and V_b. The abscissa corresponds to different values of R, while the ordinate represents the voltages V_a and V_b that are on nodes a and b for different values of R. The curves for V_a and V_b diverge for growing values of R, and V_a approaches V_{DD} while V_b approaches 0.

In order to carry out test generation and simulation for this fault, it is necessary to determine how the voltages V_a and V_b are interpreted by the inputs of the gates k and m, which are driven by a and b, respectively. The model assumes that each gate input has a threshold $\vartheta \in [0, V_{\mathrm{DD}}]$, such that voltages between 0V and ϑ are interpreted as logic 0, while voltages between ϑ and V_{DD} are interpreted as logic 1. The threshold depends on several factors like the gate type and the capacitive load on the driven line; thus, usually each input of each gate has an own but exactly defined threshold. In this example, the input of gate k which is driven by gate g has the threshold ϑ_k, which is the voltage present on node a if the resistance R of the bridge equals R_k. R_k is called a *critical resistance* because gate k interprets the voltage V_a as logic 0 if $R < R_k$ or as logic 1 if $R > R_k$. Hence, the bridge induces a wrong logic value on the input of k for $R < R_k$, while the fault remains undetected by k if the bridge's resistance is greater than R_k. At the same time, there is a critical resistance R_m (for which $V_b = \vartheta_m$) such that the input of m interprets a wrong logic value only if $R < R_m$. The intuition behind this model is that a bridge with a lower resistance allows nodes a and b to influence each other more strongly than a high-resistance bridge.

Figure 10 (b) also illustrates the situation in which the pattern 01 is applied to the inputs of gate g instead of 00. Due to the internal structure of CMOS NAND gates, the output of gate g still produces logic 1, but it will be driven with less strength, which results in a voltage V'_a which is consistently lower than V_a for all R-values. In presence of the bridge, also the intermediate voltage that is present on both a and b for $R = 0\Omega$ is lower than in the first case, which leads to a voltage V'_b which is consistently lower than V_b. The voltages V'_a and V'_b are represented by dashed curves. As can be seen in the diagram, the application of pattern 01 to gate g's inputs results in new critical resistances R'_k and R'_m that differ from R_k and R_m.

Assume that the bridge's resistance equals R_{act} in an actual circuit's instance, and that $R_k < R_{\mathrm{act}} < R_m$ (see Figure 10 (b)). Then, the input of k driven by g retains its fault-free behaviour (i.e. it interprets V_a as logic 1) while the input of m driven by h displays faulty behaviour (i.e. it interprets V_b as logic 1) under the application of 0011 to g and h. Under the application of 0111, the situation is inverted (then, $R'_m < R_{\mathrm{act}} < R'_k$): the input of k becomes fault-affected and the input of m remains fault-free. This shows that the same physical defect can behave differently at the logic level depending on the applied test pattern.

Different test generation and fault simulation tools based on this model have been developed at the University of Freiburg [183, 184, 75, 76, 78, 77, 74]. Detailed information can be also found in [179]. In addition, a similar model was also used by the author of this thesis in order to compute the realistic fault coverage of small-delay faults caused by resistive-open defects [51]. In that work, exact timing simulation is performed in order to determine detectability intervals in the timing domain, i.e. for each delay fault f and each test pair $\mathfrak{p}$, the algorithm determines the exact intervals within which the actual delay of the defect must lie in order to guarantee the detection of f by $\mathfrak{p}$. Then, the determined intervals in the timing domain are mapped to detectability intervals in the resistance domain, and the fault's overall detection probability is determined in function of the probability distribution of the open's resistance.

2.6 Fault simulation

Logic simulation is the process of determining the logic values implied on each circuit line by the application of an input pattern to the circuit.

Given a combinational circuit, the zero-delay logic simulation of an input pattern $p := b_1 \cdots b_n \in \mathbb{B}^n$ assigns each bit b_i of the pattern to the corresponding primary input and computes the new logic value implied on each line, where the lines are processed in topological order. These steps are repeated for every pattern to be simulated.

A standard technique to reduce the run-time of a simulation algorithm is called *event-driven* simulation. Given a set of patterns $P := \{p_1, \dots, p_{r_P}\}$, the first pattern is simulated as usual. For $i = 2, \dots, r_P$, values are assigned to each circuit input according to p_i. If an input's new value differs from its old value, all that input's successor gates are inserted into a priority queue that orders the contained gates topologically. Then, it suffices to recompute the logic values of the signals that are in the queue. If the value of a line taken from the queue changes, that line's successors have to be inserted into the queue as well. These steps are repeated until the queue becomes empty.

Fault simulation is the process of determining the set of faults that are detected by a given test set. Two important applications of fault simulation are the computation of fault coverage, and the combination with test pattern generation in order to avoid the generation of tests for faults that can be detected by already generated test patterns.

Algorithm 1 describes a simple fault simulation method that is independent of the used fault model. Here, C^f stands for the faulty version of the circuit when it is affected by fault f, b_l stands for the logic value on a line l and b_l^f stands for the logic value on a line l in the faulty circuit.

The fault-free simulation of a test pattern p (line 4) consists in determining the logic values that stabilise on every line of the fault-free circuit after applying p to

Algorithm 1
Simple fault simulation

```
Inputs: circuit $C$, fault list $F$, test set $P$
Output: list of detected faults $F'$
 1: SIMPLE-FAULT-SIMULATION($C, F, P$) {
 2:     $F' := \varnothing$
 3:     for each test $p \in P$ do {
 4:         SIMULATION($C, p$)                                  ▷ fault-free simulation
 5:         record computed logic value $b_z$ for every circuit output $z$
 6:         for each fault $f \in F$ do {
 7:             SIMULATION($C^f, p$)                            ▷ faulty-case simulation
 8:             if $b_z \neq b_z^f$ for any circuit output $z$ then {    ▷ if $p$ detects $f$
 9:                 move $f$ from $F$ to $F'$
10:             }
11:         }
12:     }
13:     return $F'$
14: }
```

its inputs. The implementation is fault-model-dependent. In the case of the SAFM, this simulation would correspond to the logic simulation introduced above.

Faulty-case simulation of a test pattern p (line 7) is the process of determining the logic values that stabilise on every line of the circuit when it is affected by a fault f. Usually, faulty-circuit simulation is performed using the same algorithm as for the fault-free simulation, where the fault effect is *injected* at the fault site and propagated to the circuit outputs in an event-driven manner.

Note that, in line 9, the detected fault f is not only copied to the output fault list, but also removed from the input fault list. This is done in order to avoid the unnecessary repeated simulation of a fault that has already been classified as detected. However, this is only valid when the fault model distinguishes solely between either undetected and fully detected faults. As was explained in Section 2.5, the definition of detectability is more complex for certain fault models like resistive-bridging faults or resistive opens. For instance, in [51], the detection probability of each fault is computed depending on the probability distribution for the resistance of the corresponding resistive-open defect and depending on what resistance intervals are covered by each test pattern. In such cases, every fault needs to be simulated for every test, as the detection probability of a fault is in general different for each pattern that detects it, and the overall detection probability results from the accumulated detection probabilities computed for different tests.

There are several forms of advanced fault simulation aimed at improving the runtime. These include *deductive fault simulation* [18], *concurrent fault simulation* [248], *critical-path tracing* [14] and *parallel-pattern single-fault propagation* (PPSFP) [249]. PPSFP is a method that simulates n test patterns concurrently. The logic values of each line under n different test patterns are stored in n-bit words. Then, for the evaluation of a logic gate, Boolean instructions are applied to the n-bit operands, which generates output values for all n patterns in parallel. Obviously, the combination of parallel-pattern and event-driven simulation is not trivial, because events may occur only for some of the n patterns being simulated in parallel, but implementations of event-driven PPSFP simulators with a good speed-up exist [73].

2.7 Test pattern generation

Automatic test pattern generation (ATPG) is the process of deciding whether a fault f is detectable, and of computing a test pattern that detects f if that is the case. Although the fault detection problem for combinational circuits is NP-complete [128], the average complexity of most ATPG instances found in practice is only $\mathcal{O}(n^3)$ [256, 189]. However, ATPG still remains one of the most challenging test tasks, especially due to the large number of instances that usually need to be considered. For example, in mid-sized industrial circuits with half a million gates, millions of stuck-at faults may need to be targeted depending on the average number of branches per fan-out node. In order to overcome the problem's complexity, test pattern generation is usually combined with fault simulation. After the generation of a certain number of tests, these are simulated in order to determine which not yet targeted faults are also detected by them. Faults detected by simulation can be removed from the target fault list, thus reducing the number of test generation instances to be solved. This technique is known as *fault dropping*. Another positive effect of fault dropping is the reduction of pattern count, which is of concern because the cost of test application depends strongly on the number of test patterns to be applied. In order to further reduce the number of generated tests without loss of fault coverage, *test compaction* techniques are also usually employed.

Typically, test generation processes consist of the following phases:

1. low-cost, fault-independent test generation,
2. fast identification of undetectable faults,
3. high-cost, deterministic, fault-oriented test generation,
4. static test compaction.

In Phase 1, usually random test patterns are generated and simulated in an iterative process that stops when adding more random patterns to the test set does not significantly improve the fault coverage. In Phase 2, undetectable faults are identified, e.g. through the analysis of regions between fan-out nodes and reconvergent gates [173, 172], and removed from the fault list. However, the employed algorithms are usually not complete because they are meant to be fast and low-cost, while the problem of identifying undetectable faults is co-NP-complete[4]. In Phase 3, still undetected faults are targeted by a deterministic test generation algorithm. In order to reduce the number of generated patterns, this algorithm may include *dynamic compaction* techniques, i.e. techniques that guide the test search such that each generated test is suitable for the detection of a higher number of faults. Finally, in Phase 4, *static compaction* techniques are applied in order to further reduce the size of the generated test set without loss of fault coverage.

2.7.1 Structural test pattern generation for stuck-at faults

In this section, deterministic ATPG algorithms for stuck-at faults in combinational circuits are presented. The focus is put on *structural* algorithms, as these are the first kind of ATPG algorithms to have arisen, and since they continue to be the base of ATPG tools used in industry. Structural means that these algorithms search for a solution based solely on the circuit's structure, i.e. the reasoning required to guide the search is derived directly from the gate-level net list.

The first steps in structural testing of logic circuits were made by Eldred in 1959 [72], but it was Roth's work at IBM which resulted in the first systematic ATPG method — the *D-Algorithm* [201, 202]. The algorithm allows *multiple-path sensitisation*, i.e. fault effects can be propagated over several reconvergent paths. This is an important feature, as there are faults that cannot be detected using only single-path sensitisation [216]. In fact, the D-Algorithm is *complete*, i.e. it is guaranteed to find a solution (a test pattern) if the given fault is detectable, or to prove the fault's undetectability.

Algorithm 2 describes a D-Algorithm version that gives propagation priority over justification. However, this assumption does not alter the algorithm's completeness [12]. The algorithm uses a five-valued logic known as *Roth's logic* (Table 2). The values in this logic are composite values that represent a line's logic value in the

[4] This topic is not relevant to this thesis and will not be explained in more detail. See [12, 34, 129] for more information.

Algorithm 2
The D-Algorithm

```
Inputs: circuit $C$, fault $f$
Output: returns *detectable* if $f$ is detectable, otherwise *undetectable*
D-ALGORITHM($C$,$f$) {
    set all circuit lines to x
    if $f$ is a s-a-0 fault then {
        assign D to the fault location                ▷ excite the fault
    } else {
        assign D′ to the fault location               ▷ excite the fault
    }
    return PROCEED-SEARCH( )
}

PROCEED-SEARCH( ) {
    if IMPLY-AND-CHECK( ) fails then {
        return *undetectable*
    }
    if no primary output produces an error then {
        return PROPAGATE( )
    } else {
        return JUSTIFY( )
    }
}
```

Table 2
Roth's logic [202]

	meaning	
value	fault-free case	faulty case
0	0	0
1	1	1
x	x	x
D	1	0
D′	0	1

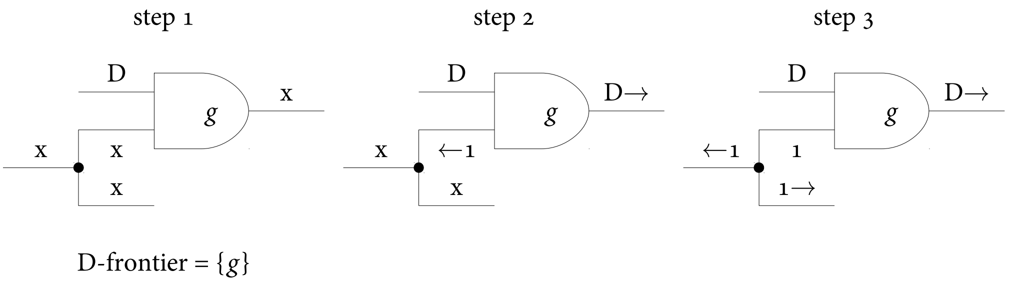


FIGURE 11. D-ALGORITHM — AN EXAMPLE CHAIN OF IMPLICATIONS

fault-free and in the faulty case. Logic operations on these values are computed by applying the operations separately to the fault-free and faulty components and by composing the results. The value x (*unspecified*) is used to represent lines that have not yet been assigned a value by the algorithm, while the values D and D′ represent errors.

First, the algorithm assigns an error value to the fault site depending on whether the fault is a stuck-at-0 or a stuck-at-1 fault. Then, the algorithm calls a recursive search function (line 8) that performs two basic tasks, *propagation* and *justification*. Propagation consists in driving the fault effect towards the primary outputs and is stopped as soon as an error can be observed on at least one primary output. Justification is a process that *justifies* values on gate outputs by values on the gate's inputs, i.e. the gate's inputs are set such that the gate produces the desired value at its output. Two data structures are used to manage propagation and justification. The *D-frontier* contains all gates through which propagation can be driven, i.e. gates whose output cannot be inferred from the current assignments, but with at least one D or D′-input. The *J-frontier* contains all gates with a specified output value, but where the current assignment of its inputs does not logically imply the value at the gate's output.

The recursive search function first calls the IMPLY-AND-CHECK-function (line 11), which computes all implications that can be deduced from current assignments without making any decisions. An example chain of implications is shown in Figure 11. The example assumes that gate *g* is the only gate in the D-frontier. Then, the need to propagate the D-value at *g*'s first input implies that *g*'s output has to be set to D and that that value has to be propagated, for which *g*'s successor gate has to be added to the D-frontier (not shown in the picture). The D-values at *g*'s first input and at *g*'s output imply a 1-value (the non-controlling value of *g*) at *g*'s second input and the need to justify that value (step 2). This justification task implies further propagation and justification tasks along the stem and second branch

Algorithm 3
Sub-routines of the D-Algorithm

```
PROPAGATE( ) {
    if D-frontier is empty then {                ▷ no further propagation posssible
        return undetectable
    }
    while D-frontier is not empty do {
        select a gate $g$ from D-frontier                              ▷ decision
        assign NCV($g$) to every unspecified input of $g$     ▷ sensitisation condition
        if PROCEED-SEARCH( ) returns detectable then {
            return detectable
        } else {
            undo last assignments                                  ▷ backtracking
        }
    }
    return undetectable                          ▷ no further propagation posssible
}

JUSTIFY( ) {
    if J-frontier is empty then {        ▷ all justification tasks have been satisfied
        return detectable
    }
    take a gate $g$ from J-frontier
    while $g$ has unspecified inputs do {
        select an unspecified input $i$ of $g$                         ▷ decision
        assign CV($g$) to $i$
        if PROCEED-SEARCH( ) returns detectable then {
            return detectable
        } else {
            assign NCV($g$) to $i$                                 ▷ backtracking
        }
    }
    return undetectable                              ▷ justification of $g$ failed
}
```

of that fan-out node (step 3). The IMPLY-AND-CHECK-function fails if the current assignments result in conflicting implications, for instance if a value is implied on a line that has been previously assigned a different specified value.

After calling the IMPLY-AND-CHECK-function, the main search function either calls the propagation or the justification sub-routine (Algorithm 3), which make propagation or justification decisions and recursively call the main search function.

Propagation consists in selecting a gate from the D-frontier and setting its unspecified inputs to the gate's non-controlling value in order to sensitise the gate to the fault effect. Since it suffices to observe a fault effect on only one primary input, usually not all gates in the D-frontier need to be processed, and the selection of a specific gate can result in a conflict. Backtracking is implemented by the while loop (line 5). If the recursive search call within the loop is unsuccessful, a new run of the while loop tries a different propagation path. Propagation fails when the D-frontier becomes empty and the fault effect is still not visible at a primary output.

Justification consists in justifying the values of the gates in the J-frontier. In contrast to the D-frontier, all gates in the J-frontier need to be processed, as all justification tasks are necessary to satisfy the propagation conditions. The only case in which the justification procedure needs to make a selection is when justifying a gate's controlling value (non-controlling value if the gate is inverting). In this case, it suffices to set only one input of the gate to its controlling value, while all other inputs may remain unspecified. If the recursive search call is unsuccessful, a new run of the while loop (line 21) tries a different input of the gate. Justification fails if no input of the gate can be set to the gate's controlling value, thus making it impossible to justify the gate's output value.

Two important ATPG algorithms that can be seen as derivatives of the D-Algorithm are *PODEM* (Path-Oriented Decision Making) [98] and *FAN* (Fan-out Oriented Test Generation) [89, 87]. In PODEM, decision making is restricted to the primary inputs, thus reducing the complexity of the algorithm to $\mathcal{O}(2^{\text{number of PIs}})$, whereas the D-Algorithm's worst-case complexity is $\mathcal{O}(2^{\text{number of lines}})$. PODEM defines *objectives* (justification tasks) to be satisfied. The first objective is given by the fault excitation condition. Further objectives are dictated by the sensitisation of gates in the D-frontier. A fast *backtracing* procedure [233] is used to select a PI assignment likely to imply the currently targeted objective. If the implications that follow from that assignment lead to no conflict, the next objective can be targeted in the same manner, and the algorithm continues to target objectives until a fault effect is visible at a primary output. If the assignment to a PI leads to a conflict, first that assignment is reverted (backtracking). However, if the reverted assignment leads to a new conflict, also previously made assignments need to be reverted. If no backtracking is possible any more, the fault is classified as undetectable.

FAN is an extension of PODEM that further improves the efficiency of structural ATPG. In contrast to PODEM, FAN allows backtracing to stop at *head lines*. Head lines are defined such that the sub-circuits driving them are fan-out-free. Hence, values on head lines can be justified without conflicting with values previously assigned to other lines in the circuit. Thus, the need for backtracking is considerably reduced, and FAN is significantly more efficient than PODEM [89]. In addition, FAN uses a *multiple-backtrace* procedure that attempts to satisfy several objectives simultaneously. Some structural algorithms implemented in commercial and proprietary ATPG tools are known to be based on FAN which is an efficient algorithm able to solve a large number of easy-to-solve ATPG instances very fast.

Most proposed enhancements of these basic ATPG algorithms [140, 218, 93, 159, 250, 108] are structural as well and rely on learning techniques in order to improve the performance of structural ATPG on hard-to-solve ATPG instances.

2.7.2 Compaction

Test compaction is the process of reducing the size of a test set without affecting the fault coverage achieved by it, thus diminishing both test application time and tester memory demand, and hence the total test application cost. Two types of compaction algorithms are known: *static* and *dynamic* techniques.

Static compaction acts on a previously generated test set and produces a smaller test set that detects at least the same faults as the original one. Some static compaction methods identify redundant test patterns using fault-simulation-based methods that are especially effective when applied to test sets containing only deterministically generated test patterns. For instance, *reverse-order fault simulation* (ROFS) [12] consists in simulating the generated test patterns in the reverse order in which they were generated. Test patterns that detect no new faults when they are simulated are dropped from the test set. After reverse-order fault simulation, also *random-order fault simulation* can be applied a number of times in order to further reduce the pattern count.

Forward-looking reverse-order fault simulation (FLROFS) [187] is an extension of ROFS. This method records which test pattern was the first to detect which fault. Since these data can be collected during the test generation process, the collection causes no significant overhead. After the test generation, normal ROFS is performed, but test patterns that have not been recorded as the first to detect any of the remaining faults can be dropped from the test set without simulation, as the test set is guaranteed to contain not yet simulated test patterns that detect those faults, namely their corresponding first test patterns. This method not only leads to smaller pattern counts than simple ROFS, but it also requires fewer simulation runs.

A further static compaction method merges pairs of test patterns into new patterns that detect at least the same faults as the original patterns. Merging is possible due to the fact that many circuit inputs are not assigned a specific logic value by the ATPG algorithm. Two *partially specified* test patterns $p_1 := b_{1,1} \cdots b_{1,n}$ and $p_2 := b_{2,1} \cdots b_{2,n}$,

$b_{i,j} \in \{0, 1, x\}$, are *compatible* if they do not assign contradicting values to any primary input, i.e. if for all $j = 1, \dots n$, either $b_{1,j} = x$ or $b_{2,j} = x$ or $b_{1,j} = b_{2,j}$. If p_1 and p_2 are compatible, they can be *merged* into a new test pattern $p_1 \cap p_2$, where the intersection operator $\cap$ is defined as in Table 3. For instance, the *intersection* of 01xx and 0x10 is 0110. Obviously, all faults detected by p_1 and by p_2 are detected by $p_1 \cap p_2$ as well. Hence, p_1 and p_2 can be replaced by only one new test pattern $p_1 \cap p_2$.

Table 3
The intersection operator

∩	0	1	x
0	0	-	0
1	-	1	1
x	0	1	x

Given a test set P, pairs of compatible tests in P are subsequently identified and replaced by their intersection until no further compaction can be achieved. The obtained compacted test set depends on the order in which the test patterns are merged. For example (taken from [12]), consider the test set $\{01x, 0x1, 0x0, x01\}$. If the first two tests are merged first, the set $\{011, 0x0, x01\}$ is obtained which cannot be compacted any further. In contrast, merging the first and the third test patterns first renders the test set $\{010, 0x1, x01\}$, which can be further compacted to $\{010, 001\}$. However, finding the optimal compaction is computationally complex. An optimal solution can be found by constructing a *compatibility graph*, i.e. an undirected graph where the nodes represent the test patterns and the edges connect compatible tests. Then, all cliques[5] contained in the graph represent sets of test patterns that are all compatible to each other and can thus be merged into one single test pattern. The optimal solution is found by covering the compatibility graph using a minimum number of cliques, which is an NP-complete problem [135].

For this reason, most static compaction algorithms have to rely on heuristic techniques for fault reordering and *test relaxation* [133, 71], i.e. the post-ATPG injection of x-values into the generated test patterns, a technique which is also widely employed to aid test compression for built-in self-test or for test of systems-on-a-chip [71].

In contrast to static compaction, which is always applied after the ATPG process has been completed and is independent of the used ATPG method, dynamic compaction

[5] A clique is a graph in which every two nodes are connected by an edge.

encompasses techniques that modify the ATPG algorithm such that each generated test pattern is suitable for the detection of a higher number of faults.

The generic approach [99] consists in generating a test pattern p_1 for a *primary target* fault f_1. Then, a *secondary target* fault f_2 is chosen, and a test pattern p_2 is generated for f_2 under the condition that the circuit inputs assigned to specified values by p_1 be assigned to the same values by p_2. Hence, if p_2 exists, it detects both f_1 and f_2 and p_1 can be dismissed. This process can be repeated for further secondary targets until the percentage of unspecified values in the test pattern is too low to allow the consideration of more targets.

Obviously, the most relevant problem is the selection of secondary target faults. Similarly to the selection of merging pairs in static compaction, the optimal selection of secondary target faults would be computationally too expensive. Hence, the selection relies on heuristic methods. For instance, the partially specified test pattern that has been generated for the primary fault can be simulated using normal fault simulation or faster algorithms like *critical-path tracing* [14], which is usually done in any case for the purpose of fault dropping. The values this simulation process implies on lines in the whole circuit are then analysed in order to determine what secondary fault is more likely to be detected by a pattern that respects these value assignments [100, 101, 13].

The pattern counts achieved by the tool COMPACTEST [185] for the ISCAS'85 [32] and ISCAS'89 [31] benchmark circuits are among the lowest ever recorded. The tool combines pre-ATPG fault reordering based on the concept of *independent faults* [17] with a more aggressive approach that allows to modify specified primary-input assignments. Also, the objectives of line justification can be changed dynamically to allow different faults to be potentially detected. However, the large number of heuristic methods used in this work makes it hard to predict the tool's performance on newer benchmark circuits, like ITC'99 [7, 48] or NXP (see Appendix A) circuits. The methods implemented in COMPACTEST were also combined with static compaction and further heuristic techniques in [134].

An alternative type of approach was presented in [170]. The *subscripted D-Algorithm* attempts to sensitise multiple paths simultaneously, thus generating a single pattern that detects many faults. However, this approach uses a system that consists of a flexible observation signal assigned to a gate's output, and of flexible control signals assigned to all gate inputs. The multiplicity of these flexible signals causes new types of conflicts that require heuristic handling [146].

Further techniques that differ from the general approach were presented in [20] and [203]. In [20], instead of extending a new generated test pattern, it is merged with a previously generated test pattern compatible to it, if one exists. Thus, the necessity to select secondary target faults is eliminated. However, like in static compaction, the selection of a previously generated pattern has a strong influence on the final result. In [203], the filling of unspecified bits in the generated test patterns is done employing a genetic algorithm[6].

[6] *Genetic algorithms* belong to the class of *evolutionary algorithms* [36, 102], which are heuristic search methods that mimic the process of natural evolution. Candidate solutions (represented by character strings) are the individuals of a population. The population evolves in an iterative process that consists in generating new individuals (derived from existing individuals by applying operations like mutation or crossover) and in selecting which individuals will form the next generation according to a fitness function that best represents the optimisation objective of the search. Evolutionary algorithms have been employed in many areas of VLSI design and test [66], including ATPG [94, 137], BIST configuration [263, 180] and test data compression [181].

3

Introduction to the SAT problem and to SAT-based ATPG

This chapter introduces the SAT problem, which is the problem of deciding whether the variables of a Boolean formula can be assigned such that the formula evaluates to logic 1. After a formal introduction of the SAT problem, the chapter presents basic algorithms for the solution of this problem, as well as the most important techniques used by modern SAT solving tools. The second part of this chapter focuses on the application of SAT solving to test pattern generation. After the introduction of the basic principle, a review of previously existing works on SAT-based ATPG is given, as well as of related ATPG approaches. Like Chapter 2, this is an introductory chapter. Hence, all its contents constitute pre-existing knowledge originated in the work of other authors, and references to the original works have been added where appropriate.

3.1 Introduction

The *Boolean satisfiability problem* (SAT) is the problem of deciding whether a Boolean formula is *satisfiable*, i.e. whether its variables can be assigned the values 0 or 1 such that the whole formula evaluates to 1. Software tools used to determine the satisfiability of SAT formulae are called *SAT solvers*.

Currently, SAT solvers are used in many fields like planning [136, 96], electronic design automation [166], and verification and test of digital systems [219, 27, 46, 224, 114, 77, 164, 69, 55, 209, 207], especially because many search problems can be converted into SAT problems efficiently [151].

The SAT problem is NP-complete [47], which means that there is currently no known method to solve an arbitrary instance of the SAT problem efficiently. However, many problems found in practice result in SAT formulae that are relatively easy to solve, especially if it is possible to nest structural information of the original problem into the SAT problem [237]. This also applies to ATPG. Although ATPG is NP-complete, only a relatively low percentage of ATPG instances found in practice make ATPG algorithms display their worst-case behaviour [256, 189, 55]. However, ATPG is challenging because it has to be applied to a very large number of instances. Thanks to techniques like *incremental SAT solving*, where the solving of a SAT instance benefits from knowledge learnt during the solution of previous SAT instances, ATPG can be performed efficiently using SAT solvers.

The first approaches to reduce the ATPG problem to a SAT problem were proposed several decades ago [220, 147, 148, 237], yet structural algorithms, which perform a direct search based on the circuit's net list, have largely remained the standard used in industrial applications due to their better run-times. Structural algorithms are particularly fast when applied to a large number of easy-to-solve ATPG instances (see also Section 2.7.1).

However, recent experiments [242, 55] have shown that *SAT-based test pattern generation* (SAT-ATPG) outperforms structural methods for hard-to-solve ATPG instances. These instances are either *hard-to-detect* faults or undetectable faults, for which structural algorithms tend to display their worst-case behaviour due to the size of the search space that needs to be traversed until a solution is found or until the nonexistence of a solution can be proved. In contrast, SAT solvers are routinely used to prove unsatisfiability in applications such as equivalence checking and model checking[7], which has given rise to numerous techniques that allow SAT solvers to prune large parts of the solution space efficiently. Obviously, this development has made SAT-ATPG more suitable than structural algorithms to prove fault undetectability. These techniques also result in an advantage of SAT-ATPG when applied to hard-to-detect faults, for which structural methods require a larger number of backtracks than SAT-ATPG.

[7] *Equivalence*, *model* and *property checking* are formal-verification problems. Given two models of the same system, or a model of the system and a specification, equivalence checking is the problem of deciding whether both models are functionally equivalent. Model and property checking refers to the problem whether a system model satisfies a certain property. For example, property checking can be used to determine whether a sequential circuit can enter a set of desired or undesired states as the result of the application of test sequences of a given length.

Especially the ability to prove undetectability is an important feature of SAT-ATPG, as modern fault modelling approaches often lead to a high number of undetectable faults, while an accurate computation of defect coverage depends strongly on the number of undetectable faults that can be reliably identified as such. For instance, the rise in importance of reliability concerns has resulted in an increased use of redundant structures to enhance fault tolerance [227, 260] (see also Chapter 9). Hence, a rising number of modelled faults are undetectable. In addition, many defects in nano-scale manufacturing technologies need to be modelled using non-standard fault models rather than the SAFM [16]. Such models usually impose very specific conditions on several lines in the circuit in order to excite the fault, thus leading to a high amount of undetectable faults. For instance, a high-resistance short defect might require particular values at the inputs of the gates that drive the bridged lines in order to justify the voltages that excite the fault (see Section 2.5). Another example is the accurate modelling of interconnect-open defects, which requires that particular values be assigned to lines that have a coupling capacitance with the affected line [204, 107, 234, 116].

Given this particular strength of SAT-based ATPG, the combination of structural methods and SAT-ATPG is an approach with expected good performance in industrial settings, as shown in [242]. But SAT-ATPG has also been proved highly useful in its own merit, especially in applications that produce many hard-to-solve ATPG instances, like the classification of faults in robust circuits with redundant checking logic [126] (see also Section 9.3), or test generation using complex fault models that require non-trivial constraints for fault activation [55, 57, 56] (see also Chapters 7 and 8).

3.2 Formal definition of the SAT problem

(Propositional) logic variables or *Boolean variables* are variables that can take either the value 0 (false) or 1 (true). In this thesis, logic variables are denoted by upper-case letters, usually X, if applicable followed by an index.

(Propositional) logic formulae or *Boolean formulae* are expressions over the set of Boolean variables and the following symbols: $\{\neg, \wedge, \vee, (,)\}$. Boolean formulae will be denoted by lower-case Greek letters.

Valid Boolean formulae are defined recursively. If X is a logic variable, then the expressions X and $\neg X$ are Boolean formulae. Also, if φ and ψ are Boolean formulae, the following expressions are Boolean formulae as well:

- $\neg\varphi$ — *negation*,
- $(\varphi \wedge \psi)$ — *conjunction*,
- $(\varphi \vee \psi)$ — *disjunction*.

For simplicity, pairs of round brackets in a Boolean formula can be omitted provided that the removal does not provoke confusion. In addition, the following abbreviating expressions are common:

- $(\varphi \oplus \psi)$ — *exclusive disjunction* — as an abbreviation of $((\varphi\wedge\neg\psi)\vee(\psi\wedge\neg\varphi))$,
- $(\varphi \rightarrow \psi)$ — *implication* — as an abbreviation of $(\neg\varphi \vee \psi)$,
- $(\varphi \leftrightarrow \psi)$ — *equivalence* — as an abbreviation of $((\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi))$,
- $(\varphi_1 \vee \varphi_2 \vee \varphi_3)$ — as an abbreviation of $((\varphi_1 \vee \varphi_2) \vee \varphi_3)$ or of $(\varphi_1 \vee (\varphi_2 \vee \varphi_3))$,
- $(\varphi_1 \wedge \varphi_2 \wedge \varphi_3)$ — as an abbreviation of $((\varphi_1 \wedge \varphi_2) \wedge \varphi_3)$ or of $(\varphi_1 \wedge (\varphi_2 \wedge \varphi_3))$.

Let $\mathfrak{X}$ be the set of variables occurring in a formula φ. A *logic assignment* or *Boolean assignment* is a map $w : \mathfrak{X} \rightarrow \mathbb{B}$ that assigns each variable either the logic value 0 or the logic value 1. If $w(X) = 1$ for a variable $X \in \mathfrak{X}$, w is said to *satisfy* X (written $w \vDash X$).

w can be extended to a map w^* that *evaluates* the formula φ. w^* is defined recursively as follows:

- $w^*(X) = w(X)$,
- $w^*(\neg\varphi) = \neg w^*(\varphi)$,
- $w^*(\varphi \wedge \psi) = w^*(\varphi) \cdot w^*(\psi)$,
- $w^*(\varphi \vee \psi) = w^*(\varphi) + w^*(\psi)$.

A formula φ is called *satisfiable* if there is an assignment w such that $w^*(\varphi) = 1$. Then, w is said to be a *model* of φ or to *satisfy* φ (written $w \vDash \varphi$). The SAT problem is the problem of deciding whether a formula φ is satisfiable.

A formula that evaluates to 0 for all assignments is called *unsatisfiable*. A formula that evaluates to 1 for all assignments is called a *tautology*.

Let $\{X_1, \ldots, X_n\}$ be the set of variables occurring in a formula φ. Then, the formula φ *describes* the Boolean function $\mathbb{B}^n \rightarrow \mathbb{B}$ that maps $(a_1, \ldots, a_n)$ to $w^*(\varphi)$ for all assignments w with $w(X_i) = a_i$ for all $i = 1, \ldots, n$.

Two formulae φ and ψ are called *semantically equivalent* if $w^*(\varphi) = w^*(\psi)$ for all assignments w, i.e. if φ and ψ describe the same Boolean function.

A formula φ is said to be in *conjunctive normal form* (CNF), if φ is a conjunction of disjunctions of literals, i.e. if it has the form $(\lambda_{1,1} \vee \cdots \vee \lambda_{1,n_1}) \wedge \cdots \wedge (\lambda_{m,1} \vee \cdots \vee \lambda_{m,n_m})$, where each $\lambda_{i,j}$ is a *literal*, i.e. $\lambda_{i,j}$ is either a variable or a negated variable[8]. For every Boolean formula there is a semantically equivalent formula in CNF. Hence, for every Boolean function $\mathbb{B}^n \to \mathbb{B}$, there is a formula in CNF that describes it.

The conjunctions that form a CNF formula are also called *clauses* and regarded as sets of literals, while the CNF formula is seen as a set of clauses. Hence, a formula in CNF is often written in the form $\{\{\lambda_{1,1}, \ldots, \lambda_{1,n_1}\}, \ldots, \{\lambda_{m,1}, \ldots, \lambda_{m,n_m}\}\}$. Clauses that consist of only one literal are called *unit clauses*.

An assignment w that satisfies a CNF formula φ has to satisfy every individual clause in φ, while each clause is satisfied if w satisfies at least one literal contained in the clause. Hence, unit clauses can only be satisfied by assignments that satisfy their only literal. Furthermore, a clause in which at least one literal occurs in both affirmative and negative form is always satisfied.

Let K_1 and K_2 be two clauses in which a literal λ occurs neither in affirmative nor in negative form. *Resolution* is an inference rule that states that if an assignment w satisfies both $K_1 \cup \{\lambda\}$ and $K_2 \cup \{\neg\lambda\}$, then w satisfies $K_1 \cup K_2$ as well. $K_1 \cup K_2$ is called the *resolvent* of $K_1 \cup \{\lambda\}$ and $K_2 \cup \{\neg\lambda\}$.

The *empty clause* is formally defined as it can be inferred by resolution. It is unsatisfiable, as it is the resolvent of $\{X\}$ and $\{\neg X\}$ for any Boolean variable X, i.e. it is implied by the conjunction of $\{X\}$ and $\{\neg X\}$, which is clearly unsatisfiable[9]. Intuitively, the empty clause is unsatisfiable because it contains no satisfied literals. In addition, the *empty formula* can be defined as well. The empty formula is satisfiable, as it contains no unsatisfied clauses.

3.3 SAT solving algorithms

3.3.1 The DPLL-Algorithm

Modern SAT solvers almost exclusively process SAT instances expressed as formulae in CNF. The use of CNF as the preferred normal form has historical reasons. The

[8] In this thesis, literals are denoted by λ, if applicable followed by an index. The expression $\neg\lambda$, which is formally not a valid logic expression, is used as an abbreviation to denote $\neg X$ if λ stands for a variable X (*affirmative occurrence*), or to denote X if λ stands for $\neg X$ (*negative occurrence*).

[9] Any Boolean formula that contains such a pair of clauses is called *self-contradictory* and is unsatisfiable.

SAT solving algorithm presented by Davis and Putnam in 1960 [60] was intended to efficiently determine the satisfiability of a sequence of SAT instances that arose from the attempt to show the unsatisfiability of a *first-order predicate logic* formula.

Given an infinite sequence of Boolean formulae $\varphi_1, \varphi_2, \varphi_3, \ldots$, a natural number n was to be found, for which $\psi_n := \varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_n$ would be unsatisfiable. Hence, it was necessary to subsequently determine the satisfiability of $\psi_1, \psi_2, \ldots$. The decision to transform all φ_i into CNF expressions allowed the fast construction of the ψ_j. Each ψ_j is constructed by simply concatenating ψ_{j-1} and φ_j and is thus automatically in CNF.

The algorithm by Davis and Putnam consists in iteratively modifying the input formula φ by applying one of three rules until either the modified formula becomes empty, in which case φ is satisfiable, or until the empty clause is inferred, in which case φ is unsatisfiable. The three rules are as follows:

- *Unit propagation* — If a unit clause $\{\lambda\}$ occurs in the formula, delete all clauses that contain λ, and delete all occurrences of $\neg\lambda$ in the remaining clauses.
- *Pure literal* — If the formula contains a *pure literal* λ, i.e. if $\neg\lambda$ occurs in no clause, delete all clauses that contain λ.
- *Resolution* — Choose two clauses K_1 and K_2 such that K_1 contains a literal λ and K_2 contains $\neg\lambda$. Then, delete K_1 and K_2 and add the resolvent of K_1 and K_2 to the formula.

In 1962, Davis, Logemann and Loveland replaced the resolution rule by a depth-first search with backtracking, thus eliminating the first algorithm's high memory demand, and also allowing the algorithm to not just prove satisfiability, but to derive a model as well. The resulting algorithm is known as *DLL* or *DPLL-Algorithm* [59].

Also algorithms not based on DPLL have been proposed, for instance Bryant's work with BDDs[10] [33] or Stålmarck's Proof Procedure [223]. Although these algorithms perform well on small SAT instances, their further development has stagnated, and

[10] Boolean functions can be represented using *binary decision diagrams* (BDD), i.e. directed acyclic graphs where the nodes represent Boolean variables and the edges represent logic assignments to the variables at whose node they originate. The nodes without outgoing edges are called leafs and stand for the logic value to which the formula evaluates if the variables are assigned the values specified by the edges on the path from the root node to the leaf. An important property of reduced BDDs is that, given a fixed variable ordering, they are a canonical representation of the Boolean function. However, due to their high memory demand, BDDs are impractical for Boolean functions with a large number of variables.

Algorithm 4
The DPLL-Algorithm

```
Inputs: Boolean formula $\varphi$ in CNF
Output: returns satisfiable if $\varphi$ is satisfiable, otherwise unsatisfiable
 1: DPLL($\varphi$) {
 2:     for each unit clause $K$ in $\varphi$ do {
 3:         $\varphi$ := APPLY-UNIT-PROPAGATION($\varphi, K$)
 4:     }
 5:     for each pure literal $\lambda$ in $\varphi$ do {
 6:         $\varphi$ := APPLY-PURE-LITERAL-RULE($\varphi, \lambda$)
 7:     }
 8:     if $\varphi$ is empty then {
 9:         return satisfiable
10:     }
11:     if $\varphi$ contains the empty clause then {
12:         return unsatisfiable
13:     }
14:     $X$ := SELECT-VARIABLE($\varphi$)
15:     if DPLL($\varphi \wedge X$) returns satisfiable then {    ▷ decision ($X = 1$)
16:         return satisfiable
17:     } else {
18:         return DPLL($\varphi \wedge \neg X$)                  ▷ backtracking
19:     }
20: }
```

DPLL has become the foundation of modern high-performance SAT solvers [151], including the SAT solvers MiraXT [152] and ANTOM [217], which have been used by the author of this thesis as SAT solving back-end engines for SAT-based ATPG.

The DPLL procedure is shown in Algorithm 4. The procedure consists of three parts. The first part (lines 2–7) is characterised by the assignment of necessary values to certain Boolean variables. In order to satisfy the input formula, pure literals and unit-clause literals that occur in affirmative form have to be mapped to logic 1, while negative literals need to be mapped to logic 0. Since the application of the unit propagation and pure-literal rules deletes clauses and literals from clauses, the second part of the DPLL-Algorithm (lines 8–13) checks whether a clause or the formula itself have become empty, in which case the algorithm terminates. Otherwise, the third part of the algorithm consisting of the depth-first search is applied. First, the SELECT-VARIABLE-procedure (line 14) selects a *decision variable* according to some criterion. Then, the algorithm maps that variable to logic 1 and recursively calls the DPLL-procedure, which will determine the satisfiability of the derived formula that results from that assignment. If that formula is not satisfiable, the chosen variable is set to logic 0 (backtracking) and the satisfiability of the new derived formula is checked.

If the formula is satisfiable, the assignments made during the application of the unit propagation and pure-literal rules and during the search dictate the found model. However, the model is not unique and depends on the order in which decision variables are selected. In addition, a particular selection can result in a model that is only partially specified, as the clauses deleted by the application of the unit propagation and pure-literal rules can contain variables that are completely eliminated from the formula before being selected as decision variables.

One of the most important aspects for the performance of the DPLL-Algorithm is the strategy by which the decision variables are chosen [151]. In [59], for instance, a simple heuristic reordering was reported to cause a speed-up by a factor of 10. This is still one of the key issues in modern SAT solvers, and a good method for the selection of decision variables can have a large influence on the solver's performance.

3.3.2 Modern SAT solvers

The growing complexity of SAT instances derived from problems in various fields of engineering resulted in several SAT solvers being introduced towards the end of the 1990s. Examples include *GRASP* [163, 165], *SATO* [258], *rel_sat* [26] and *WalkSAT* [221, 168]. Essentially, these solvers combine heuristic techniques for local search with simplified implementations of the DPLL-Algorithm that result in better run-time and memory efficiency. For instance, while the original DPLL-Algorithm generates validity proofs (a list of all resolution steps), modern SAT solvers are purely search-based and do not physically delete unsatisfied variables in order to avoid run-time-expensive operations.

The most significant advancements were contributed by the solver *Chaff* [176, 259] which enhanced some of the techniques first implemented in GRASP and SATO. In particular, Chaff introduced the *VSIDS* strategy (Variable-State Independent Decaying Sum), which has been proved to be a very good technique for the selection of decision variables. The advanced techniques implemented in Chaff are still employed by SAT solvers of today, though they have been extended and enhanced by SAT solvers like *BerkMin* [103], *MiniSat* [80, 79, 1], *MiraXT* [152] and ANTOM [217].

The run-time efficiency of modern SAT solvers is characterised by four main aspects that will be discussed in more detail in the remainder of this section: pre-processing, efficient Boolean constraint propagation, learning and the use of good decision strategies.

Pre-processing encompasses several techniques employed to eliminate trivial variables and clauses from the SAT formula prior to the start of the DPLL-based search algorithm. The necessity of pre-processing arises from the fact that real-world SAT instances are derived from other problems in engineering. The translation of such problems into SAT formulae needs to be run-time-efficient and often results in SAT formulae that contain many redundant variables and clauses. Examples of pre-processing techniques include the application of unit propagation and of the pure-literal rule, as well as techniques introduced by MiniSat based on subsumption[11] and variable elimination through resolution. The extent to which pre-processing is applied varies among different SAT solvers, and the result depends on the complexity of the SAT formula as well. For instance, the pure-literal rule is often not applied as the effort spent on identifying pure literals may not be compensated by the lower complexity of the resulting simplified formula.

Boolean constraint propagation (BCP) is the process of computing all logic values that are implied by the current partial assignment of variables. BCP is called after the SAT solver has made a decision. Since BCP consumes between 70 and 95% of the total SAT solving time [151], this is one of the most optimised procedures in the solver. The most important contribution to the efficiency of BCP was Chaff's *watched-literals scheme* which exploits the fact that not every clause containing the decision variable needs to be examined after a decision, because no clause that is already satisfied or that contains at least two unspecified literals can trigger an immediate implication of the assignment made to the decision variable. In this scheme, the solver "watches" only two unspecified literals of each clause. Then, a clause needs to be examined only if one of its two watched literals becomes unsatisfied. This allows implications to be found quickly while examining only a small amount of clauses.

Another major step in the advancement of modern SAT solvers was the introduction of *conflict analysis* in Grasp. Conflict analysis is done using *non-chronological backtracking* and *recording of conflict clauses* (also known as *learning*). The solver manages an implication graph that shows what implications are forced by each clause. When a conflict is encountered by BCP, the implication graph is used to find the first reason for the conflict, called the *first unique implication point* (first UIP). Then, backtracking can skip several decision levels and directly jump back to

[11] A clause K_1 *subsumes* a clause K_2 if $K_1 \subseteq K_2$. K_2 can be removed from the SAT formula, as every model of K_1 satisfies K_2 as well.

that point. Also, a conflict clause is resolved from the path between the UIP and the conflict and added to the clause database. This learnt clause will prevent the SAT solver from making the same chain of decisions again, thus effectively limiting the search space. However, a large amount of learnt clauses can not only result in memory explosion, but also considerably slow down BCP. In practice, thousands of clauses are learnt every second. Hence, while Grasp considers multiple UIPs, Chaff considers only the first UIP in order to limit the number of learnt clauses and in order to learn shorter clauses which truncate larger parts of the search space. Also, older conflict clauses can be deleted after some time. BerkMin introduced a concept where conflict clauses are learnt based on their *activity*, i.e. clauses that are more often involved in conflicts are seen as more useful and kept for a longer time.

Finally, *decision strategies* are strategies by which decision variables are chosen, and they have a big influence on the solver's performance. In general, decision strategies try to choose variables whose specification will constraint the SAT problem such that large parts of the solution space can be disregarded quickly. For instance, Marques-Silva introduced four strategies based on literal counts: *DLCS* (Dynamic Largest Combined Sum), *DLIS* (Dynamic Largest Individual Sum), *RDLCS* (Random DLCS) and *RDLIS* (Random DLIS) [162]. These strategies select variables depending on their number of occurrences in the formula and assign these variables either random values or a value depending on the difference in number between the variable's affirmative and negative occurrences. A drawback of these strategies is that the literal counts need to be constantly updated. In an attempt to overcome this difficulty, Chaff introduced the VSIDS (Variable-State Independent Decaying Sum) strategy that works as follows: The number of affirmative and negative occurrences of each literal in the original formula are counted and kept in a sorted list. These counters are incremented only when new learnt clauses are added. After the addition of a given number of new learnt clauses, all counters are divided by two (*decay operations*) and the list is resorted. Then, whenever a variable needs to be chosen, the variable with the largest affirmative or negative occurrence is chosen and assigned the logic value 1 if the affirmative occurrences outnumber the negative ones, or 0 if the negative occurrences outnumber the affirmative ones (as in DLIS). Aside from the significant run-time reduction that arises from the more efficient counting, VSIDS's periodical halving of counters gives precedence to variables that occur in recently learnt clauses, i.e. to variables often involved in conflicts, which results in an intuitively more intelligent search algorithm. The VSIDS strategy and its variants are now used by most SAT solvers.

3.3.3 Incremental SAT solving

Learning has also been beneficial for the solution of the *incremental Boolean satisfiability problem* (incremental SAT). The formulation and solution of this problem was motivated by work on SAT-based circuit verification [121] and defined in [120] as the problem of deciding the satisfiability of a CNF formula $\varphi \cup \{K\}$ for a clause K, given φ's satisfiability. Although the problem is NP-complete, it can be solved more efficiently once the satisfiability of φ has been determined. The algorithm proposed in [120] solves the SAT instance φ using the DPLL-Algorithm and uses the information contained in the built search tree to speed up the solution of $\varphi \cup \{K\}$. The extended problem of deciding the satisfiability of a series of *n related* SAT formulae $\varphi \cup \psi_1, \ldots, \varphi \cup \psi_n$, where φ is a shared prefix formula, was addressed with the introduction of the tool *SATIRE* [255]. SATIRE uses GRASP's learning mechanism such that the solving of $\varphi \cup \psi_{i+1}$ benefits from the conflicts learnt during the solving of $\varphi \cup \psi_1, \ldots, \varphi \cup \psi_i$, for $i = 1, \ldots, n - 1$. However, learning is adapted such that only conflict clauses pertaining to the prefix φ are kept in the clause database. The improvement of learning techniques as well as better approaches to identify such conflict clauses [226, 81] allows modern SAT solvers to solve the incremental SAT problem efficiently.

3.3.4 SAT solving with qualitative preferences

In many applications, having a satisfying assignment is not enough. In planning, for instance, a SAT solution corresponds to merely one plan, while an optimal plan is a plan that satisfies a set of additional soft goals. Approaches to extend SAT in order to specify problems that require more expressive power, while still benefiting from the advancements in conventional SAT solving, include MIN-ONE[12] and MAX-SAT[13] [50, 230], DISTANCE-SAT[14] [21] and *pseudo-Boolean* approaches [24].

Giunchiglia and Maratea [95, 96] proposed a mechanism known as *SAT solving with qualitative preferences*, where the SAT solver is given a list of variables that should preferably be assigned to 1. An advantage of this approach is that it is possible to extend a SAT solver to handle preferences in an efficient way [65].

[12] *MIN-ONE* is the problem of finding a model that assigns 1 to a minimum number of variables.

[13] *MAX-SAT* is the problem of finding a model that satisfies a maximum number of clauses.

[14] *DISTANCE-SAT* is the problem of finding a model that assigns at most d variables differently from a given partial assignment that stands for a solution preference.

Let φ be a Boolean formula, and let $\mathfrak{L}$ be the set of all literals occurring in φ. A *qualitative preference* on $\mathfrak{L}$ is a subset $L \subseteq \mathfrak{L}$ together with a partial ordering $\prec$ on L. Intuitively, L is the set of literals that should be preferably satisfied, and $\prec$ determines the relative importance of those preferences.

Let w_1 and w_2 be two models of φ. Formally, w_1 is *preferred* over w_2 under $(L, \prec)$ (written $w_1 \prec_L w_2$) if the following two conditions hold:

- There is at least one literal $\lambda \in L$ that is satisfied by w_1 but not by w_2 (written $w_1 \prec_\lambda w_2$).
- If there is a literal $\lambda \in L$ with $w_2 \prec_\lambda w_1$, then there is a literal $\lambda' \in L$ with $w_1 \prec_{\lambda'} w_2$ and $\lambda' \prec \lambda$.

This mechanism also allows to formally define the optimality of a model. A model w is an *optimal model* of φ under $(L, \prec)$, if $w \prec_L w'$ for every other model w' of φ.

The following example (taken from [65]) shall illustrate what type of problems can be expressed using this formalism. The CNF formula

$$\{\{\neg \mathit{fish}, \neg \mathit{beef}\}, \{\neg \mathit{redwine}, \neg \mathit{whitewine}\}\}$$

models the fact that one can have neither both fish and beef nor both red and white wine. In order to express that one would like to have fish and both red and white wine, but white rather than red wine if having both was not possible, the SAT problem is extended by preferences $(\{\mathit{fish}, \mathit{whitewine}, \mathit{redwine}\}, \mathit{whitewine} \prec \mathit{redwine})$. These preferences instruct the SAT solver to search for the formula's only optimal model $(\mathit{fish}, \mathit{beef}, \mathit{redwine}, \mathit{whitewine}) \mapsto (1, 0, 0, 1)$.

3.4 The principle of SAT-based ATPG

Boolean difference was the first method that analysed errors in logic circuits by converting the original problem into a SAT problem. It was published in 1968 [220].

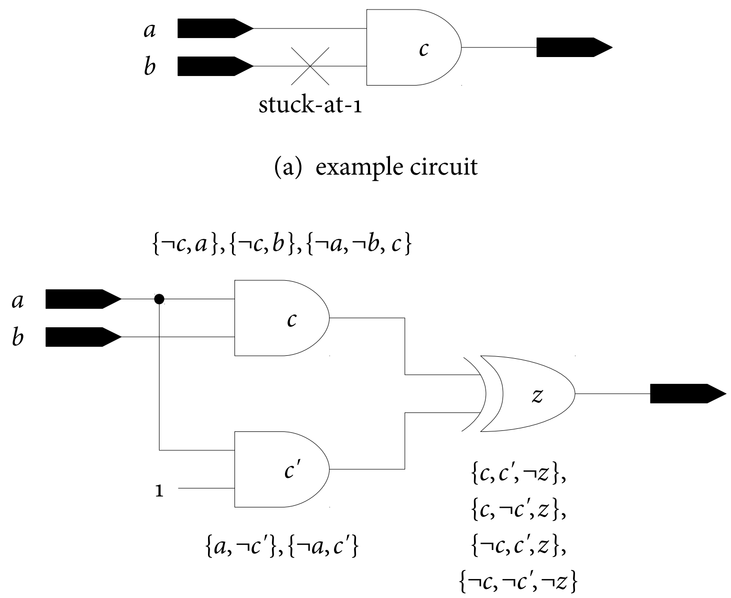
Given a combinational circuit C with n inputs and m outputs, SAT-based test generation for a fault f consists of three steps: First, fault f is injected into the original circuit, which renders a faulty version C^f of the circuit. Then, the original circuit and the faulty version are combined to form a *miter* [29], i.e. a circuit with n inputs and one output z, where the i-th input of C and the i-th input of C^f are both connected to the miter's i-th input for $i = 1, \ldots, n$, while the j-th output of C and the j-th output of C^f are both connected to a new XOR gate for $j = 1, \ldots, m$. The outputs of all these new XOR gates are connected to an m-input tree of OR gates, whose output is connected to z. An input pattern can induce the logic value 1 on z if and only if the responses of C and C^f differ on at least one output, i.e. if the input pattern detects f. Hence, finding a satisfying assignment for the Boolean function implemented by the miter under the condition that the miter's output be assigned to 1 is equivalent to generating a test pattern for fault f, while the nonexistence of a satisfying assignment proves the fault's undetectability.

The second step consists in generating a SAT instance in CNF that represents the ATPG problem. Every line in the circuit is represented by a Boolean variable. Then, the SAT instance is composed of clauses that describe the circuit's structure, as well as of a one-literal clause that can only be satisfied by assignments that satisfy the Boolean variable that represents the miter's outputs. The clauses that describe the circuit's structure are generated using Tseitin transformation [246], a transformation method that has the advantage that both the number of clauses it generates as well as its run-time are only linear in the number of gates.

Finally, the last step consists in solving the generated SAT instance using a SAT solver. If the SAT solver finds a solution, the test pattern is composed of the values that have been assigned to the Boolean variables that represent the primary inputs.

Consider, for instance, test generation for the stuck-at-1 fault at line b of the circuit shown in Figure 12 (a). The corresponding miter is shown in Figure 12 (b). Here, the AND gate c' represents the faulty version of the original circuit. Since the original circuit has only one output, the tree of OR gates is unnecessary and the miter's output is given by the XOR gate's output z.

The functionality of gate c is described by the Boolean expression $c \leftrightarrow (a \wedge b)$, which is an abbreviation of $(\neg c \vee (a \wedge b)) \wedge (\neg(a \wedge b) \vee c)$. By the axioms of the Boolean algebra (see Section 2.1), this expression is equivalent to $(\neg c \vee a) \wedge (\neg c \vee b) \wedge (\neg a \vee \neg b \vee c)$,



(a) example circuit

(b) corresponding miter

FIGURE 12. MITER CONSTRUCTION AND CONVERSION INTO A SAT FORMULA

which is an expression in CNF that can be written in the form

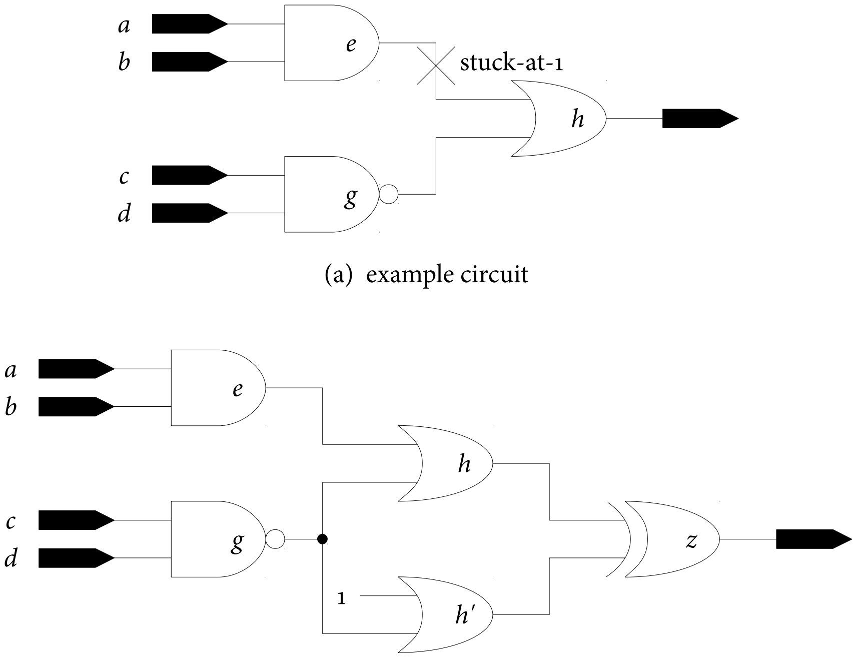
$$\{\{\neg c, a\}, \{\neg c, b\}, \{\neg a, \neg b, c\}\}.$$

This type of transformation can also be applied to the XOR gate and to the second AND gate c', which acts like a buffer due to its second output being a constant logic 1. The complete SAT instance is composed of all clauses shown in Figure 12 (b), together with the clause $\{z\}$ that requires that any satisfying assignment assign the miter's output to 1. The only model of this SAT instance is given by

$$(a, b, c, c', z) \mapsto (1, 0, 0, 1, 1).$$

The extracted test pattern is composed of the values assigned to a and b, i.e. 10.

Important improvements of this basic problem formulation were published in the first half of the 1990s. In [147, 148], instead of duplicating the whole circuit, only the sub-circuit that is affected by the modelled fault is duplicated in order to reduce the size of the generated SAT instance. This is illustrated in Figure 13. In this example, only gate h needs to be duplicated (h'), as that is the only gate affected by the fault. The sub-circuit that drives gate h's faulty input requires no duplication, as that input's behaviour in the faulty case depends only on the definition of the fault.



(a) example circuit

(b) corresponding miter

Figure 13. Larrabee's miter

The part of the circuit that drives the second input of gate h (gate g and its input cone) is also not affected by the fault's presence; hence, it can be shared with h'.

Another improvement that has become a standard in SAT-ATPG was implemented in the tool *TEGUS* (Test Generation Using Satisfiability) [237]. The employed technique includes structural information into the SAT instance, which guides the search of the SAT solver. Inspired by the D-Algorithm, which only tries to drive propagation through gates in the D-frontier, this method explicitly models *D-chains*, i.e. paths along which propagation can be driven.

In this approach, all lines in the fault site's output cone are modelled not only by two Boolean variables, one that represents the line's logic value in the fault-free case, and one that represents the line's logic value in the faulty case, but also by an additional Boolean variable that models whether the line would have a D or a D′-value in the D-Algorithm. Let X_1 and X_2 be the variables that represent the fault-free and faulty logic value of a given line. Let X_3 be the new variable. Then, all models of the set of clauses $\{\{X_3, \neg X_1, X_2\}, \{X_3, X_1, \neg X_2\}, \{\neg X_3, X_1, X_2\}, \{\neg X_3, \neg X_1, \neg X_2\}\}$ assign X_3 the value 1 if and only if X_1 and X_2 are assigned different values, i.e. if the modelled line displays a fault effect. Hence, this set of clauses is added to the original SAT instance. Finally, all new variables along each D-chain are connected by additional clauses that model the fact that if a gate g belongs to a D-chain (i.e. its D-chain variable is set to 1), then one of its successors must belong to a D-chain as well in order to enable the propagation of the fault along a path starting at g.

Although these new variables and clauses increase the size of the SAT-instance, the run-time required for SAT solving is considerably reduced, as corroborated by the implementation of these technique in other tools [224, 55]. The reason for this is that a considerable number of backtracks are avoided during the SAT solving due to the implications that are triggered by the new clauses. In addition, thanks to the D-chain variables assigned to the primary outputs in the fault site's output cone, the miter's array of XOR gates does not need to be modelled explicitly any more.

3.5 Previous and related work

Even after the optimisations proposed in [147, 148, 237], which also included the use of global implications and other techniques derived from structural ATPG, the efficiency achieved by algorithms like PODEM and FAN for the average ATPG instance could not be transferred to early SAT-based approaches, as the SAT solver is not able to identify Boolean variables that represent primary inputs or head lines and to make decisions based on them.

In response to this, alternative approaches were developed, which attempted to enhance the performance of basic structural ATPG by combination with graph-based algorithms instead of SAT. For instance, in [235], BDDs were used to enhance justification and propagation. However, the worst-case memory complexity of BDDs is exponential, which makes them inapplicable to large circuits, and BDD-based techniques always compute all possible justifications even when only one is needed, which results in over-specified test patterns that cannot be compacted well.

A different approach that attempts to combine Boolean and structural reasoning in one model utilises *implication graphs*, which are also partially used in [148]. This approach was implemented in the tool *IGRAINE* (Implication-GRaph-bAsed engINE) [240, 238, 239]. An implication graph is a directed acyclic graph, whose nodes represent assignments to lines and whose edges represent implications. There is also a second type of nodes, called $\wedge$-nodes, which are used to represent ternary relations between line assignments. Graph algorithms are employed to derive indirect implications that would remain undetected in purely structural ATPG, and the method has the advantage that its memory complexity is only linear in the number of gates. Furthermore, these indirect implications can be used to aid the constraint propagation in SAT solving. [97] proposes the use of new data structures for a better representation of k-nary relations for arbitrary k-values. This technique was implemented in the tool *SPIRIT* (Satisfiability Problem Implementation for Redundancy Identification and Test generation), but the presented data structures suffer from a large overhead when applied to gates with many inputs.

Recently, the advancement of research on efficient SAT solving made after the year 2000 has given rise to a new generation of purely SAT-based approaches. The most relevant contributions were made by a research group at the University of Bremen between 2005 and 2010 [69], and by the author of this thesis at the Chair of Computer Architecture at the University of Freiburg between 2008 and 2012.

The tool PASSAT [224], which uses the SAT solver MiniSat [80, 1], constitutes the first contribution by the University of Bremen, followed by comparative studies with NXP's structural ATPG tool Amsal [5] and a flow that combines both tools [242, 67]. [68] presents an extension of PASSAT aimed at increasing the amount of unspecified bits in the generated patterns, which is an important prerequisite for better static compaction (Section 2.7.2).

In Bremen, research towards enhancing the run-time efficiency of SAT-ATPG developed along two paths. The first approach uses BDDs to generate smaller SAT instances and to also reduce the run-time needed for SAT instance generation [244, 243]. The circuit is partitioned into FFRs, and each FFR is represented using a BDD. Then, the SAT instance that describes the structure of the circuit is derived from the BDDs, which removes some information redundancy and allows to reuse already converted sub-formulae. However, this approach is only applicable to FFRs with a limited amount of gates due to the memory requirements of BDDs.

The second path followed in Bremen, which involves approaches that utilise learning and incremental SAT solving [86, 70, 241], lead to better results. For instance, in [70], a central database is used to cache basic as well as learnt clauses that represent the circuit in the fault-free case. Since not all cached clauses are needed in all SAT instances, these are activated or deactivated dynamically. This approach reduces the time needed for SAT solving as well as the time needed to generate the SAT instances.

The contributions made by the author of this thesis constitute the main topic of this work and are discussed in detail in Chapters 4–9.

4

The SAT-based test pattern generator Tiguan

This chapter introduces the SAT-based test pattern generator Tiguan. First, a summary of the tool's development history is given. Then, in order to allow for an accurate description of Tiguan's core algorithms, the conditional multiple stuck-at fault model (CMS@FM) is first briefly defined. This is the fault model used by Tiguan for the internal representation of faults. A thorough motivation for the need for such a fault model will be given in Chapter 7 which is dedicated to the solution of ATPG problems for complex fault models and of ATPG problems with optimisation goals. The rest of the chapter discusses the basic implementation of Tiguan, which uses MiraXT as SAT solving back-end, and presents a first evaluation of the tool's performance.

Author's contribution — The author's contribution consisted in the implementation of a new, stand-alone SAT-ATPG framework (Tiguan), the definition of a generic fault model (CMS@) to enable the framework's applicability to different types of test generation problems, and a refinement of the initial implementation in order to make the new tool competitive in comparison to the then-state-of-the-art SAT-ATPG tool PASSAT developed at the University of Bremen.

Parts of the work covered in this chapter have been published in [J2, C16, W10] (see author's publications on pages 223–226).

4.1 Introduction

From this chapter onwards, this thesis covers the contributions to the field of SAT-based test pattern generation made by the thesis's author between 2008 and 2012. The beginning of the work consisted in implementing a new SAT-ATPG tool from scratch, where the main concern was to create a particularly efficient implementation that would permit the incorporation of new techniques to increase run-time efficiency. The resulting tool Tiguan (**T**hread-parallel **I**ntegrated test pattern **G**enerator **U**tilising satisfiability **AN**alysis) was presented for the first time in [54], and a journal version of this work was published in [55]. All subsequent methods developed by the author have been integrated into this tool. Furthermore, Tiguan has been used as a SAT-ATPG back-end engine in several works [126, 131, 52, 82, 83], and its software code base has served as basis for spin-off tools developed by other doctoral students at the Chair of Computer Architecture in Freiburg [208, 206, 209, 211, 205, 210, 207].

The SAT solving back-end engine used in Tiguan's initial implementation was the SAT solver *MiraXT* [152, 151] (also developed at the Chair of Computer Architecture in Freiburg), a state-of-the-art SAT solver that incorporates various optimisation techniques. Furthermore, MiraXT supports thread parallelism, which permits the use of multi-processor systems or multi-core processors to full capacity. A tight integration of both tools allows Tiguan to dynamically control internal parameters of the SAT solver, for instance whether the SAT formula is to be pre-processed prior to solving, or what number of computation threads to use, and reduces run-time and memory requirements as clauses can be passed to the SAT solver's running instance without the need of expensive communication via the file system.

MiraXT's support of thread parallelism was used to explore manners of improving the run-time efficiency of SAT-ATPG for single stuck-at faults. A *two-stage method* was developed, where faults are processed using different SAT solving parameters depending on the hardness of the produced SAT instances, thus enabling the utilisation of parallelism in an effective way.

Tiguan also incorporates a pattern-parallel fault-simulator [73], which was, like MiraXT, also developed at the Chair of Computer Architecture in Freiburg. Thus, Tiguan is a complete ATPG framework able to use simulation for fault dropping (see Section 2.7) and for compaction-aiding techniques like reverse-order fault simulation (see Section 2.7.2).

The combination of all these factors allowed Tiguan to classify, without aborts, all stuck-at faults in nineteen large industrial circuits containing up to two and a half million gates, which were provided by NXP Semiconductors GmbH Hamburg [5][15], whereas the processing of several hard-to-solve instances was aborted by a commercial ATPG tool using structural ATPG, even after the tool's backtracking limit was raised to very high levels.

After this successful work on stuck-at faults, TIGUAN was extended such as to support complex fault models, motivated by the search for new challenges to SAT-ATPG. This lead to the introduction of the *conditional multiple stuck-at fault model* (CMS@FM) [54, 55], which allows to model defects with fault effects on multiple *victim* lines that are activated simultaneously if certain conditions on a number of *aggressor* lines are satisfied. The implementation of a flexible and extensible framework to support this fault model allowed to generate test patterns for resistive bridges (Section 2.5) and gate-exhaustive testing [169, 43] with few or no aborts for circuits of various sizes.

TIGUAN also provides a mode in which the percentage of unspecified bits in the generated patterns is maximised, which is essential for static and dynamic compaction (Section 2.7.2) as well as for test compression [145, 192]. However, although high densities of unspecified bits can be measured for industrial circuits, the density is usually rather low within the input cone of the fault site. This is because SAT-ATPG searches for a test pattern based on the SAT engine's decision strategies which are not guaranteed to reflect reasoning made based on the circuit's structure. In contrast, structural ATPG methods usually search for a pattern starting at the fault site and moving towards the circuit's inputs, thus assigning only inputs that are essential for fault detection.

This motivated the next research goal — the incorporation of dynamic compaction techniques into SAT-ATPG [53]. A tight integration of these techniques into TIGUAN allows to guide the search of the SAT solver such as to generate patterns that detect several faults. TIGUAN's specific data structures and the flexibility offered by the CMS@ framework are used to dynamically extend the definition of a fault according to the conditions that are necessary to guarantee the detection of other faults.

After this work, large parts of TIGUAN's implementation were optimised in order to provide a C++ library that allowed other researchers to use TIGUAN as a SAT-ATPG back-end for various applications. Along with this, also sequential expansion over an arbitrary number of time frames (see Section 2.2.3) was implemented. This

[15] Details on these circuits and on all other benchmark circuits mentioned in this thesis are listed in Appendix A.

allows to map dynamic faults that affect the circuit's behaviour over a given number of clock cycles to CMS@ faults in the combinational expansion of the circuit.

Furthermore, a new SAT solver developed at the Chair of Computer Architecture in Freiburg was integrated into TIGUAN. The development of the SAT solver ANTOM [217] was started in 2010 with the intention of implementing a highly efficient SAT solver that incorporates the newest advances in SAT solving. The solver has been successfully deployed into different test and verification tools [117, 209, 210, 207], also as a #SAT[16] solver [82, 83], and as the foundation of a QBF[17] solver [212, 213]. Furthermore, ANTOM supports incremental SAT solving (see Section 3.3.3) and SAT solving with *assumptions*. Assumptions is the term for an initial partial assignment of Boolean variables that can be passed to the SAT solver. The solver preserves those assignments and continues the solving based on the implications of those initial assignments. These features were used to develop a new *fault clustering* technique, thanks to which the time required to classify all stuck-at faults in NXP's suite of industrial benchmarks was reduced by up to 65.3% [57]. In order to increase ANTOM's performance on ATPG instances, its internal control variables were adjusted using a dedicated parameter tuning tool [127, 3].

Moreover, ANTOM is able to solve SAT problems with qualitative preferences (see Section 3.3.4). This made it possible to extend TIGUAN to support even more complex fault models. In [57], an extension of the CMS@FM, the *enhanced conditional multiple-stuck-at fault model* (ECMS@FM), was presented. This model can specify multiple fault locations along with a set of *hard* conditions imposed on arbitrary lines; hard conditions must hold in order for the fault effect to become active. Additionally, *optimisation constraints* that may be required for best coverage can be specified via a set of *soft* conditions. The number of satisfied soft conditions can be manipulated (e.g. maximised) using SAT solving with preferences. Finally, in [56], an application of the ECMS@FM to ATPG for power droop test was presented.

The rest of this chapter introduces the CMS@FM, an example application of that fault model, and the basic SAT-ATPG algorithm for CMS@ faults implemented in TIGUAN. The next chapter focuses on the run-time efficiency of SAT-ATPG for stuck-at faults. It gives an evaluation of TIGUAN's performance on multi-core architectures based on the utilisation of MiraXT's thread parallelism, and presents the fault clustering technique that uses ANTOM's incremental SAT solving. Chapter 6 presents the work on dynamic compaction for SAT-based ATPG. Chapter 7 focuses on the application of TIGUAN to complex fault models by mapping these models to

[16] *#SAT* is the problem of determining the number of assignments that satisfy a Boolean formula.

[17] *QBF* is the problem of determining the satisfiability of a quantified Boolean formula.

CMS@ or ECMS@ faults, and on the implementation of the SAT-ATPG algorithm for ECMS@ faults. Chapter 8 discusses the use of the ECMS@ fault model to generate test patterns for power droop testing. Finally, Chapter 9 introduces the Tiguan library, the C++ library that was implemented in order to allow client applications to incorporate Tiguan's functionality. Furthermore, the chapter presents example applications of this library, which are related to the important topics of process variations and fault tolerance.

4.2 The CMS@ fault model

The basic implementation of the SAT-ATPG tool Tiguan uses the *conditional multiple stuck-at fault model* (CMS@FM) which can be seen as an extension of the SAFM, and which is related to generic fault modelling approaches such as *fault tuples* [64], the *generalised fault model* [144] or Mentor Graphics's recently introduced *user-defined fault models* [158]. Chapter 7 discusses in detail the motivation for the definition of this fault model and of its extension (the ECMS@FM), as well as their applications.

4.2.1 Definition

A CMS@ fault (A, V) is described by two sets: the set of *aggressor lines* $A := \{a_1, \dots, a_r\}$, which may be empty, and the set of *victim lines* $V := \{v_1, \dots, v_s\}$. Each aggressor line a_i is associated to a fixed logical value b_{a_i}, and each victim line v_j has a fixed stuck-at behaviour s-a-b_{v_j}. The behaviour of a circuit with a CMS@ fault is as follows. If an input vector sets each aggressor line a_i to its associated value b_{a_i}, then the fault is activated and each victim line displays its s-a-b_{v_j} behaviour.

A test pattern that detects such a fault needs to set all aggressors to their corresponding values, to excite at least one victim line v_j (i.e. set its fault-free value to $\neg b_{v_j}$), and to propagate the error produced on at least one of the excited victim lines to a primary output.

For better legibility, a CMS@ fault $(\{a_1, \dots, a_r\}, \{v_1, \dots, v_s\})$, where each aggressor line a_i is associated to the value b_{a_i}, and each victim line v_j is associated to the behaviour s-a-b_{v_j}, is written as follows:

$$\text{if } [a_1 = b_{a_1}, \dots, a_r = b_{a_r}]\ v_1 \text{ s-a-} b_{v_1}, \dots, v_s \text{ s-a-} b_{v_s}.$$

If the aggressor set is empty, the fault is written as follows: v_1 s-a-$b_{v_1}, \dots, v_s$ s-a-b_{v_s}.

The SAFM is a subset of the CMS@FM. A single-stuck-at fault is represented by a CMS@ fault consisting of an empty aggressor set and one single victim.

4.2.2 Example application to gate-exhaustive testing

Gate-exhaustive testing is an approach shown to be effective in identifying hard-to-detect defects on actual manufactured silicon [43]. Similarly to *n-detection* [186, 38] approaches, where each fault f is tested using n different patterns such as to increase the coverage of physical defects that behave like f, gate-exhaustive testing targets each stuck-at fault at the output of a gate g and applies all input combinations to g which excite the fault [169, 43]. For example, given a two-input AND gate, the stuck-at-1 fault at the gate's output is tested independently by three different patterns: one that justifies 00 at the gate's inputs, one that justifies 01, and one that justifies 10, while the stuck-at-0 fault at the gate's output is tested by only one pattern that justifies 11 at the gate's inputs.

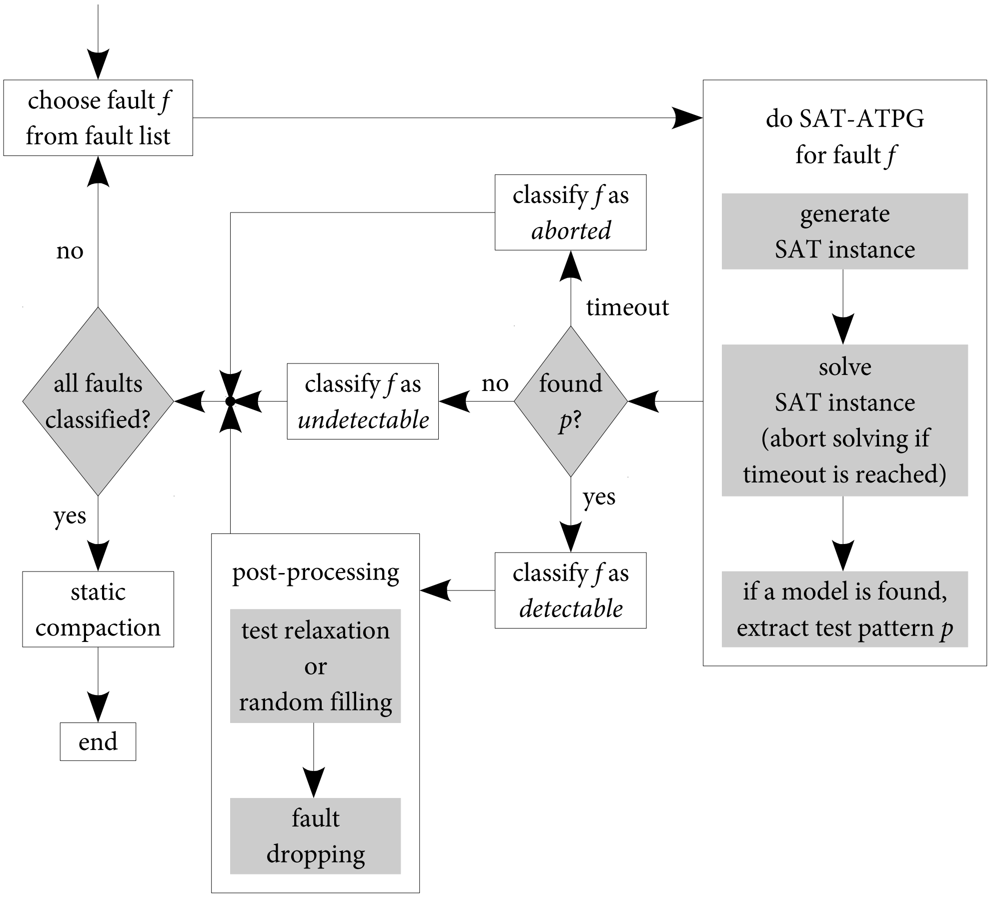
In general, $2^n - 1$ test patterns must be generated for the stuck-at-1 fault at the output of an n-input AND or NOR gate, and for the stuck-at-0 fault at the output of a NAND or OR gate. The $2^n - 1$ patterns must justify all input combinations in which at least one of the gate's inputs is set to its controlling value. For the opposite stuck-at fault, one pattern must be generated. That pattern has to set all inputs of the gate to the gate's non-controlling value, as that is the only possible way to excite the fault. For n-input XOR or XNOR gates, 2^{n-1} patterns must be generated for the stuck-at-1 fault, and another 2^{n-1} patterns must be generated for the stuck-at-0 fault at the gate's output.

Gate-exhaustive testing is easily mapped to the CMS@FM. Each targeted input combination to the given gate is expressed by declaring the gate's inputs as aggressors and imposing on them the required justification targets. For example, let g be a two-input AND gate with inputs a and b. Then, the four CMS@ faults that need to be targeted in order to exhaustively test g are the following:

- if $[a = 0, b = 0]$ g s-a-1,
- if $[a = 0, b = 1]$ g s-a-1,
- if $[a = 1, b = 0]$ g s-a-1, and
- if $[a = 1, b = 1]$ g s-a-0.

4.3 TIGUAN — OVERVIEW

Given a combinational circuit modelled at gate level, a list of CMS@ faults and a set of parameters that includes a SAT solving timeout, TIGUAN generates a test set that detects all faults whose SAT instances can be solved within the allotted time budget. Other parameters control what pre- and post-processing steps to execute, as well as internals of the SAT solving back-end. Figure 14 gives an overview of the basic flow.



choose fault *f* from fault list
do SAT-ATPG for fault *f*
generate SAT instance
solve SAT instance (abort solving if timeout is reached)
if a model is found, extract test pattern *p*
classify *f* as *aborted*
timeout
found *p*?
no
classify *f* as *undetectable*
yes
classify *f* as *detectable*
post-processing
test relaxation or random filling
fault dropping
all faults classified?
no
yes
static compaction
end

FIGURE 14. TIGUAN — FLOW

First, TIGUAN selects a fault from the input fault list. Although not shown in the figure, TIGUAN also allows to pre-sort the fault list prior to the start of the process. The order in which faults are processed can influence the total run-time when fault dropping is performed. Once a fault *f* has been selected, the SAT-ATPG engine attempts to generate a pattern for f by generating a SAT formula in CNF and passing it to the SAT solving engine. In this thesis, the process of converting the ATPG problem into a SAT formula is referred to as *SAT instance formulation* or, for conciseness, as *SAT formulation*, while the process of determining the satisfiability of the generated formula is called *SAT solving*.

The SAT solver is allotted a fixed amount of time to solve the SAT instance. If the SAT instance cannot be solved within that time, the SAT solving is aborted and a timeout is raised. If the SAT solver finds a model within the allotted time, a test pattern p that detects f is extracted from the model. The test pattern corresponds to the values assigned to the Boolean variables that represent the circuit's primary inputs. If the SAT instance is unsatisfiable, the fault is provably undetectable.

Depending on the result returned by the SAT solver, fault f is classified as detectable, undetectable or aborted. If the fault is found to be detectable, fault dropping is combined with other post-processing steps. In fault dropping, all yet-undetected faults are simulated when new patterns are generated, and faults detected by simulation are classified as detected and excluded from further processing, thus saving the run-time necessary to formulate and solve their corresponding SAT instances.

Which post-processing steps are executed together with fault dropping is controlled via the set of parameters passed to Tiguan at the beginning and can sensibly influence the performance of the tool. For example, filling the unspecified bits of the generated test pattern p with random values and performing fault simulation with the filled test pattern usually results in a larger amount of faults being detected by simulation, which leads to fewer test generation runs and thus to better run-times. In this case, however, the final test set is composed of fully specified test patterns and hence not suitable for static compaction or compression. In contrast, the test relaxation of p allows to achieve a better compaction, but fewer faults are detected by simulation when only partially specified patterns are used.

The full process terminates when all faults in the input fault list have been classified. In a last post-processing stage, static compaction can be applied to the final test set.

4.4 Generation of SAT formulae

This section explains how Tiguan generates a SAT formula φ for a given CMS@ fault f such that f is detectable if and only if φ is satisfiable.

Similarly to Larrabee's work [147, 148], where only the sub-circuit affected by the fault is duplicated to construct a simplified miter (see Section 3.4), in Tiguan only parts of the circuit that are relevant for the justification and propagation are modelled. This results in smaller SAT formulae that can be generated in a shorter time without compromising the completeness of test generation. Moreover, since the SAT problem's worst-case complexity is exponential in the number of variables, the size of the SAT formula has a big influence on the time needed for SAT solving. Depending on the structure of the circuit and the location of a stuck-at fault, the influence region of the affected gate can comprise less than 10% of the circuit's total area. Experiments with first, unpublished Tiguan prototypes showed that modelling the whole circuit and modelling only the fault site's influence region can result in run-time differences of several orders of magnitude. In that experiment, the average SAT solving time per instance measured for iscas'85 circuit c6288 went down from 10 minutes to only a few seconds.

However, in order to make the process of formulating the SAT instance as efficient as possible, Tiguan does not actually construct the reduced miter before applying the Tseitin transformation. Tiguan constructs a tree-like data structure in which every node corresponds to a gate and every edge corresponds to a connection between gates. The data structure is designed such as to enable efficient forwards (from PIs to POs) and backwards (from POs to PIs) traversing of the circuit. For example, fan-out nodes are not modelled in this data structure. Instead, nodes are allowed to have more than one outgoing edge. While information relevant to the gate is stored in the node object, information relevant to the connection between two specific gates is stored in the corresponding edge object. Traversing efficiency is essential, because the assignment of Boolean variables to lines and the formulation of clauses is done on the fly while traversing the circuit. Additionally, in order to reduce the memory requirements, formulated clauses are immediately passed to the SAT solver instead of being kept in a SAT-solver-external clause database.

Each node contains all information relevant to the gate represented by the node. In order to increase the efficiency of SAT formulation, data that do not depend on the specific fault being processed are written into the nodes only once, while only data that depend on the specific ATPG instance (e.g. the Boolean variables assigned to each site) are updated for each instance. For example, the inversion and controlling and non-controlling values of each gate need to be derived from the gate's type only once. Once these values have been recorded for each gate, the Tseitin transformation of all multiple-input gates can be handled by one single generic function that does not need to query the gate types any more.

A procedure called *colouring* is performed in order to determine which lines of the circuit need to be modelled, and to what extent. This is illustrated in Figure 15 which shows an example CMS@ fault consisting of two victim lines v_1 and v_2 and

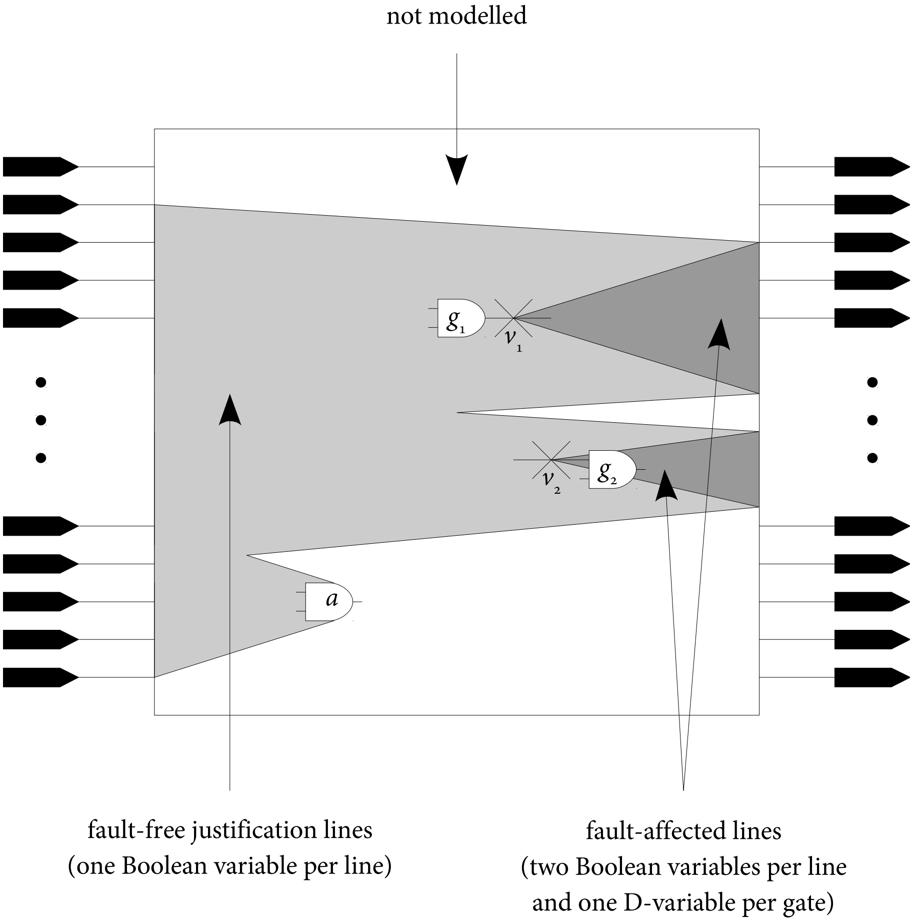

not modelled
g_1
v_1
v_2
g_2
a
fault-free justification lines
(one Boolean variable per line)
fault-affected lines
(two Boolean variables per line
and one D-variable per gate)

FIGURE 15. CIRCUIT COLOURING

one aggressor line a. First, all lines that belong to the output cone of any of the victim lines are marked as *fault-affected* (dark-grey region in Figure 15). These are lines whose logic value depends on the fault site's value and may display erroneous behaviour in the presence of the fault. In order to permit the distinction between stuck-at faults on different branches of the same fan-out node, victim lines are associated to gate ports instead of to gates. Hence, the fault-affected attribute is assigned to gate ports. In this example, the output port of gate g_1 is victim line v_1 and the first input port of gate g_2 is victim line v_2. Note that if a victim is a gate's input port (e.g. v_2), then only the victim and the gate's output are regarded as fault-affected, not the whole gate and in particular not all other inputs of the gate.

Once all fault-affected primary outputs have been determined, all lines that belong

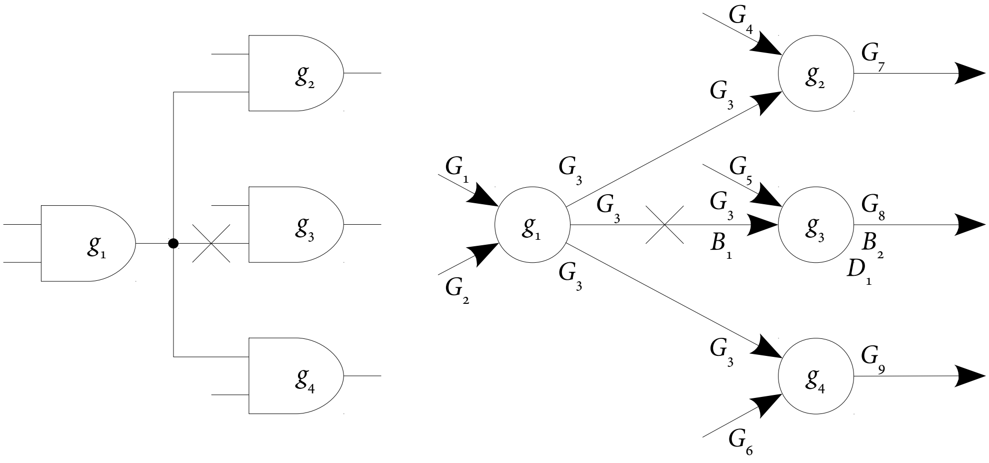


Figure 16. Assignment of Boolean variables

to the input cone of any fault-affected PO or to the input cone of any aggressor line are marked as *justification lines* (light-grey region in Figure 15). This set does not include the lines that have already been coloured dark grey. The set of justification lines comprises all lines that might have an effect on the logic value of aggressor lines or on the non-fault-affected inputs of gates with fault-affected outputs. Hence, these lines determine the justifiability of the CMS@ fault's activation condition and the propagability of observed faulty behaviour. Note that, in contrast to victim lines, aggressor lines are associated to gate outputs only.

Once the circuit colouring is finished, Boolean variables are assigned to all coloured lines in the circuit in one single pass. Lines that have been left uncoloured are excluded from the model, as such lines have no effect on the processing of the current fault. Justification lines are modelled by only one Boolean variable each. This variable represents the logic value of the line in the good circuit and is denoted by *G* followed by an index (*G* stands for *good*). Each fault-affected line is represented by two Boolean variables, a G-variable representing the line's value in the good circuit and a *B-variable* representing the line's value in the faulty version of the circuit (*B* stands for *bad*). Additionally, fault-affected gate outputs are assigned a third variable — the *D-variable* which is used to implement the *D-chain* technique (see Section 3.4) introduced in [237].

Figure 16 illustrates the assignment of Boolean variables. The circuit portion shown on the left-hand side is internally modelled using the data structure shown on the right-hand side. The cross on the second input of gate g_3 marks a victim site. In the right-hand model, variables are not assigned to edges, but rather to their source and sink points. Thus, since the later generation of clauses is node-local, all Boolean variables related to each node are directly accessible by the node object without the need of further look-up. However, the algorithm reuses Boolean variables whenever possible in order to reduce the number of assigned variables — and thus the number of clauses — to a minimum. For example, the source and sink points of all edges starting at node g_1 are assigned one single G-variable G_3, as all these gate ports will always have the same logic value in the fault-free circuit. Moreover, this strategy eliminates the need to generate clauses that express the equivalence of variables assigned to different fan-out branches.

The case of g_3 illustrates that not all input ports of a gate with a fault-affected output need to be assigned faulty-case variables. During the generation of clauses, the application of the Tseitin transformation will detect which variables need to be related to each other in order to describe the functionality of the gate. In this example, the set of clauses $\{\{\neg G_8, G_5\}, \{\neg G_8, G_3\}, \{\neg G_5, \neg G_3, G_8\}\}$ describes g_3's functionality in the fault-free case, while the set of clauses $\{\{\neg B_2, G_5\}, \{\neg B_2, B_1\}, \{\neg G_5, \neg B_1, B_2\}\}$ describes the gate's functionality in the faulty-circuit case.

When all Boolean variables have been assigned, the generation of clauses is applied locally to each node in topological order. For each node, clauses that describe its functionality are generated using the Tseitin transformation as in the example above. Additionally, for nodes whose output ports have been assigned a D-variable, clauses are generated which implement a variant of the D-chain technique which requires only one clause per gate to connect the D-variables into chains. Given a gate with input D-variables $D_1, \ldots, D_n$ and an output D-variable D_{n+1}, the generated clause is $\{\neg D_{n+1}, D_1, D_2, \ldots, D_n\}$. This clause is equivalent to the Boolean expression $D_{n+1} \rightarrow (D_1 \vee D_2 \vee \cdots \vee D_n)$. Hence, it expresses that if the output of the gate carries a fault effect, then the fault effect must also be present on at least one of its fault-affected inputs. Like in the original technique, further clauses express that a line's D-variable is equivalent to the exclusive disjunction of that line's G and B-variables.

The final component of the SAT formula is the set of *triggering clauses*, which consists of the clauses that activate the fault, the clauses that excite the fault and the clauses that require models of the SAT instance to set the miter's output to 1. Let G_a be the G-variable assigned to an aggressor line a that needs to be set to a logic value b_a in order to activate the fault. Then, if $b_a = 0$, the clause $\{\neg G_a\}$ is added to the SAT formula. Otherwise, the clause $\{G_a\}$ is added to the SAT formula. This is repeated for every aggressor line, thus setting up the conditions for fault activation.

The condition for fault excitation, namely that a fault effect must be observed on at least one victim site, is expressed by only one clause. Let $v_1, \ldots, v_n$ be the victim lines, let $G_1, \ldots, G_n$ be the G-variables assigned to those lines, and let s-a-$b_1, \ldots,$ s-a-b_n be their stuck-at behaviour. Then, the clause that is added to the SAT formula is $\{\lambda_1, \ldots, \lambda_n\}$, where λ_i stands for G_i if $b_i = 0$, or for $\neg G_i$ if $b_i = 1$, for $i = 1, \ldots, n$.

The last triggering clause is the one that represents the miter's XOR gates and tree of OR gates. Since the D-variables assigned to the modelled POs already describe the functionality of the miter's XOR gates, these gates do not need to be modelled explicitly. Also, the tree of OR gates can modelled by one single large clause $\{D_1, \ldots, D_n\}$, where the D_i are the D-variables assigned to the primary outputs.

4.5 Post-processing

Post-processing is applied to each generated test pattern individually. When combined with fault dropping, Tiguan usually replaces the unspecified bits of the generated test pattern with random specified values (*random filling*), such that more patterns are detected by simulation, thus reducing the number of necessary SAT-ATPG runs and speeding up the whole process. However, if the main goal is the generation of a compact test set, Tiguan is also able to perform test relaxation, i.e. to inject x-values into the generated test patterns without loss of fault coverage.

Trivially, all primary inputs that are not modelled in the SAT instance are left unspecified. Hence, depending on the amount of circuit area covered by the fault's influence region (coloured region), already a high percentage of PIs are unspecified just by exclusion from the model. However, a large number of PIs in the coloured region remain which can be relaxed, because fault effects are usually propagated over only a limited number of paths rather than over all fault-affected paths. This especially applies to CMS@ faults with multiple victims, as it suffices to propagate the faulty effects of only one victim in order to detect the fault.

Hence, after the generation of each test pattern, Tiguan performs an input-output-cone analysis based on which paths are sensitised by the found pattern. An example is illustrated in Figure 17. Here, the same CMS@ fault as in Figure 15 is shown, but after a test pattern p has been generated. The example assumes that p excites only victim v_1, and that fault effects are propagated only over three paths π_1, π_2 and π_3. Then, Tiguan determines the number of primary inputs that belong to the input cone of each of these paths and chooses to consider only the path whose input cone contains the smallest number of primary inputs (π_2 in the example). All primary inputs that lie outside of the chosen path's input cone can be relaxed without invalidating fault detection, however without the exclusion of PIs belonging to the input cones of aggressor lines. Also in the case that the found sensitised paths

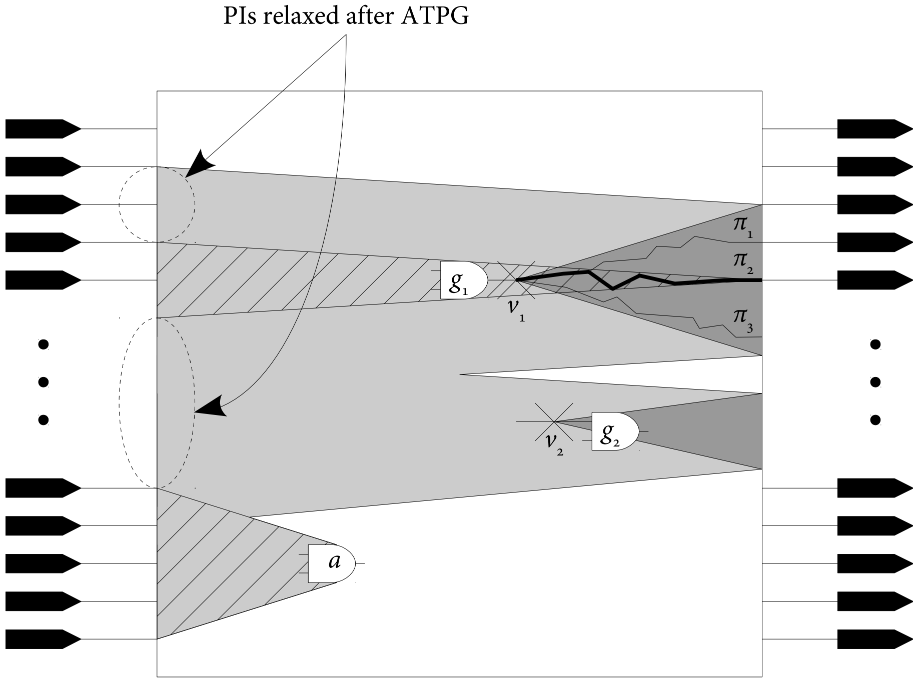


Figure 17. Input-output-cone analysis for test relaxation

belong to different victims, the path whose input cone has the smallest number of PIs is chosen, regardless of which of the victims it belongs to, as it suffices to observe the fault effect of only one victim. An advantage of SAT-ATPG is that neither simulation nor a dedicated search are required to perform this analysis. The sensitised paths correspond to primary outputs whose D-variables have been assigned to logic 1 by the SAT solver, and this information is already contained in the solution of the SAT problem. Also, the analysis of input cones of the primary outputs needs to be done only once after the initialisation of the circuit model. After the generation of each test pattern, this does not need to be repeated, as it is already known which primary outputs require the specification of less primary inputs.

Additional analysis can be carried out in order to inject more x-values into the remaining cones. In [68], a technique consisting in locally analysing justification conditions in order to relax more cones is implemented. For instance, given an AND gate whose output is 0, only one of the inputs which has been set to 0 needs to be justified. All other inputs of the gate can remain unjustified; hence the PIs in their respective input cones can be relaxed. However, this technique can be very time-consuming when combined with SAT-ATPG, as SAT-ATPG does not have additional information to aid such an analysis, whereas structural ATPG algorithms collect that kind of information without negative effect on their performance. Structural algorithms may even be optimised in a way that such an analysis is made superfluous. For example, in [185], PODEM's backtracing is modified such as to sensitise different paths every time that a value is to be justified on a specific line, thus increasing the efficiency of fault dropping. Such approaches, however, are not directly applicable to SAT-ATPG, as the main advantage of SAT-ATPG, especially when applied to undetectable and hard-to-detect instances, stems from the liberty the SAT solver has to apply SAT-specific pruning techniques that are not aware of the structure of the original problem.

One attempted approach to utilise SAT-specific techniques consisted in modifying Tiguan's SAT solving engine MiraXT, which usually returns only fully specified SAT solutions. An experimental modification allowed the relaxation of the SAT solutions to be carried out by the SAT solver instead of externally. However, the percentage of variables that could be relaxed without compromising MiraXT's runtime was very low and, in all observed cases, none of the relaxed Boolean variables corresponded to primary inputs but to internal lines of the circuit. The reason why this experiment failed is that the core SAT solving algorithm could not be changed without affecting the performance of the SAT solver; hence, the relaxation was performed by a simple and fast heuristic analysis after the normal solving routine had terminated. As was explained in Section 3.3, modern SAT solvers employ techniques that avoid repeated and thorough inspection of the clause database. The watched-literals technique is one such strategy. A further technique that has made modern solvers very efficient [151] consists in setting all variables of a SAT instance to either a fixed value or to a random value prior to the start of the search routine. Then, the search routine is aimed at solving the existing conflicts rather than at satisfying each clause individually. Such a solving paradigm is obviously counterproductive towards the generation of SAT models with a high number of unspecified variables.

The tool used for fault dropping is a 64-bit PPSFP (parallel-pattern single-fault propagation, see Section 2.6) fault simulator [73] that was originally developed for the simulation of resistive bridges [197, 75, 78] and was adapted to the CMS@FM. In order to benefit from the simulator's parallelism, fault simulation is not invoked after every SAT-ATPG run, but only when a previously fixed number of new test patterns have been accumulated. The number of patterns that are collected before doing actual simulation is called *simulation width*, and can be chosen to be any number less than 64, since the tool is not restricted to 64-bit fault simulation.

Initially, the strategy to choose the best simulation width was expected to be of concern, given that the simulator's parallel-processing capability can be utilised optimally only if the simulation width is as high as the host processor's bit width. In this case, though, waiting until enough patterns have been collected may result in the generation and solution of SAT instances corresponding to faults that would have been detected by simulation if the simulation had been carried out instantly. However, the experiments later showed that the fault simulator is so efficient that its run-time is nearly negligible in comparison to the ATPG run-time. Hence, the choice of a simulation width can be made independently of run-time concerns. Given the run-times of the individual tasks performed by Tiguan, it was determined that it is generally safe to choose a high simulation width.

The only post-processing step executed after the generation of all test patterns has finished is the static compaction, which is only effective if no random filling has been applied. The static compaction implemented in Tiguan consists of a greedy merging algorithm. Given a sequence of test patterns $p_1, \ldots, p_r$, the algorithm tests the compatibility of each pattern p_i to each pattern p_j with $j > i$ and, if found compatible, p_i is replaced by $p_i \cap p_j$, and p_j is dropped from the test set.

4.6 Evaluation of Tiguan's performance

Tiguan was applied to iscas'85 circuits and to the combinational cores of iscas'89 and itc'99 circuits, and to the suite of industrial circuit benchmarks provided by NXP. When not otherwise stated, the measurements were performed on a 2.8 GHz AMD Opteron computer with 16 GB RAM. Note that MiraXT is a 32-bit application; hence, the memory use of a Tiguan process was limited to 4 GB.

All the experiments presented in this chapter were performed without employing the SAT engine's multi-threading mode, i.e. the SAT solving of each instance was done by only one computation thread on one processor. This permitted a fair comparison to the tool developed at the University of Bremen, PASSAT, which is not capable of multi-threading, and to a commercial ATPG tool that employs a structural algorithm. An evaluation of the impact of multi-threaded SAT solving is given in Chapter 5.

4.6.1 Stuck-at faults

First, tests for stuck-at faults were generated for larger iscas'85 circuits and for the combinational cores of larger iscas'89 and itc'99 circuits. No timeout was imposed

Table 4

SAT-ATPG without fault dropping for stuck-at faults — ISCAS and ITC'99 circuits

				run-time (s)		
		classification		average per fault		
circuit	faults	detected	undetectable	formulation	solving	total
c1355	1,574	1,566	8	0.0010	0.0011	4.5
c1908	1,879	1,870	9	0.0010	0.0006	4.6
c2670	2,747	2,630	117	0.0010	0.0003	5.4
c3540	3,428	3,291	137	0.0020	0.0006	14.0
c5315	5,350	5,291	59	0.0010	0.0002	10.3
c6288	7,744	7,710	34	0.0030	0.0014	61.8
c7552	7,550	7,419	131	0.0010	0.0003	19.4
cs01196	1,242	1,242	0	0.0000	0.0001	1.0
cs01238	1,355	1,286	69	0.0000	0.0001	1.0
cs01423	1,515	1,501	14	0.0000	0.0001	1.2
cs01488	1,486	1,486	0	0.0000	0.0001	0.6
cs01494	1,506	1,494	12	0.0000	0.0001	0.6
cs05378	4,603	4,563	40	0.0000	0.0001	4.1
cs09234	6,927	6,475	452	0.0010	0.0003	16.8
cs13207	9,815	9,664	151	0.0010	0.0003	19.2
cs15850	11,725	11,336	389	0.0020	0.0006	47.8
cs35932	39,094	35,110	3,984	0.0010	0.0002	62.3
cs38417	31,180	31,015	165	0.0020	0.0003	89.7
cs38584	36,303	34,797	1,506	0.0010	0.0002	71.1
b13c	801	801	0	0.0000	0.0000	0.1
b14c	16,167	16,137	30	0.0030	0.0019	122.1
b15c	21,282	20,545	737	0.0050	0.0068	378.8
b17c	68,207	66,552	1,655	0.0050	0.0064	1,168.6
b18c	206,812	206,430	382	0.0110	0.0049	5,022.8
b20c	35,731	35,661	70	0.0050	0.0030	470.3
b21c	36,058	35,976	82	0.0050	0.0030	453.0
b22c	51,341	51,243	98	0.0050	0.0030	619.9

on SAT solving; hence, all faults were classified. Also, no fault dropping was utilised such as to allow for a realistic evaluation of the core test generation algorithm.

Table 4 reports the results of this experiment. The first two columns quote the circuit names and the number of targeted faults. The input fault lists comprised all stuck-at faults after the removal of locally equivalent faults (see Section 6.3 for details). The third column indicates how many faults were classified as detected, i.e. for which faults a test pattern could be generated (the SAT formula was found to be satisfiable), and the fourth column shows the number of provably undetectable faults, i.e. of

Table 5

SAT-ATPG with 32-bit fault dropping for stuck-at faults — NXP circuits

						run-time (s)			
		classification				average per fault			
circuit	faults	det	und	abr	patterns	frm	slv	sim	total
p35k	67,733	66,721	1,012	–	11,536	0.0330	0.0278	0.0007	1,364
p45k	68,760	68,564	196	–	3,604	0.0050	0.0017	0.0008	47
p77k	120,348	113,049	7,299	–	5,318	0.0290	0.3455	0.0510	5,454
p78k	163,310	163,310	0	–	468	0.0050	0.0006	0.0061	7
p81k	204,174	202,981	1,193	–	7,529	0.0100	0.0017	0.0015	162
p89k	150,538	148,604	1,934	–	9,868	0.0070	0.0015	0.0018	154
p100k	162,129	161,404	725	–	5,142	0.0060	0.0032	0.0028	91
p141k	282,428	279,189	3,239	–	8,893	0.0500	0.0337	0.0024	1,706
p267k	366,871	365,423	1,448	–	11,579	0.0200	0.0031	0.0037	447
p269k	369,055	367,607	1,448	–	11,633	0.0180	0.0031	0.0046	436
p286k	650,368	640,103	10,264	1	20,243	0.0410	0.0490	0.0062	3,456
p295k	472,022	468,174	3,847	1	22,786	0.0240	0.0053	0.0042	1,159
p330k	540,758	535,070	5,656	32	23,392	0.0380	0.0388	0.0048	3,208
p378k	816,534	816,534	0	–	1,107	0.0220	0.0007	0.0145	44
p388k	881,417	876,750	4,665	2	11,975	0.0290	0.0078	0.0065	830
p469k	142,751	140,869	1,762	120	578	0.0940	4.4455	1.7238	13,139
p951k	1,557,914	1,542,633	15,281	–	20,899	0.0600	0.0011	0.0119	2,668
p1522k	1,697,662	1,681,874	15,788	–	63,549	0.0730	0.0099	0.0173	9,324
p2927k	3,527,607	3,412,613	114,907	87	39,842	0.1560	0.0308	0.0602	33,758

faults whose corresponding SAT formula was found to be unsatisfiable. The next two columns quote the average time in seconds that was needed to formulate a SAT instance (*formulation*) and to solve it (*solving*). Since no fault simulation was employed, the SAT formulation and SAT solving processes were called once for each fault. The number of generated patterns is not quoted, as no compaction techniques were employed. Hence, the number of generated patterns equals the number of detected faults. Finally, the last column shows the total time in seconds that was required to process all faults. With the exception of one circuit (b18c), Tiguan was able to classify all faults with an average processing time of less than 0.01 seconds per fault. This also includes structurally complex circuits like c6288, which is a 16-bit multiplier known among the testing community as a hard benchmark.

Tiguan was also applied to large industrial circuits provided by NXP. As usual for circuits of their size, fault dropping was utilised. Fault dropping was combined with random filling to maximise the amount of faults detected by simulation, and a simulation width of 32 was chosen, i.e. 32 new generated patterns were collected before each simulation run, and all collected patterns were simulated in PPSFP fashion. The results are summarised in Table 5. Like in the previous table, the first two columns quote the circuit names and the number of targeted faults, again all stuck-at faults after the removal of equivalent faults. The third, fourth and fifth columns indicate the distribution of the faults into classes of detected (*det*), provably undetectable (*und*) and aborted (*abr*) faults. Detected faults are faults for which a test pattern could be generated or faults that were detected by simulation. Provably undetectable faults are faults whose corresponding SAT formula was found to be unsatisfiable, and aborted faults are those whose corresponding SAT formula's satisfiability could not be determined within 20 seconds. The timeout of 20 seconds per fault was chosen taking the tool PASSAT as reference, which also employed that time limit in the experiments shown later in this section. For thirteen of these nineteen circuits, Tiguan was able to classify all faults in limited SAT solving time. From the six circuits in which faults were classified as aborted, only two circuits have a number of aborts that stands out among the rest. In the case of p2927k, this can be attributed to the size of the circuit. Since the number of both variables and clauses is linear in the number of gates contained in the coloured area of the circuit, and given p2927k's large size (2,539,052 gates)[18], the SAT formulae generated for this circuit are on average larger than the formulae generated for other circuits. Hence, the same timeout of 20 seconds per fault has a greater effect on this circuit. Still, the 87 aborts constitute only 0.002% of the targeted faults. In the case of p469k, the number of aborts is higher, especially in comparison to this circuit's relatively small size (49,771 gates). However, extensive experimental data have shown that this circuit generally makes different testing tools (e.g. the stand-alone RBF simulator SUPERB [74]) display their worst-case behaviour. No exhaustive analysis has been carried out in this regard, but a possible explanation might be this circuit's large depth[19] in relation to its number of inputs and outputs. This has also been observed, though to a smaller scale, for circuit p77k. Also, the fraction of undetectable faults (i.e. of potentially harder SAT instances) in these two circuits is higher than in all other circuits.

The sixth column shows the number of test patterns that were generated, which is smaller than the number of detected faults due to the possibility of detection by simulation. Note that no dynamic or static compaction techniques were employed in the experiments presented in this chapter.

[18] See Appendix A for details on the used benchmark circuits.

[19] A circuit's *depth* is the number of gates that form the circuit's longest complete path between a primary or secondary input and a primary or secondary output.

The last four columns report the run-times measured for this experiment. All times are given in seconds. Columns *frm*, *slv* and *sim* list the average time per run required for the different stages of fault processing, i.e. for SAT formulation, SAT solving and fault simulation, respectively. The differences between these average times are best appreciated in Figure 18, which shows these data in graphical form[20]. The total time required for the complete processing of all faults is shown in the last column.

It has been noted before, e.g. in [108], that sophisticated performance enhancements are usually effective only for few hard-to-detect faults, while the processing of easy-to-detect faults tends to be slowed down. The same could be observed for TIGUAN in preliminary experiments, given that the average SAT solving time per instance is of less than 0.1 seconds for all but two of these circuits despite their large sizes. For this reason, some of MiraXT's pre-processing steps were switched off permanently, and this applies to all experimental results presented in this thesis. More details on the pre-processing techniques employed by MiraXT are given in Section 5.1.1.

The average run-times measured for the single steps of the test generation process lead to two important observations. The first observation is that, due to the efficient implementation of the employed fault simulator, the time needed for fault simulation is very low and can be regarded as nearly negligible. This is not directly evident in the table, where the quoted average simulation time per run is generally smaller than the average SAT solving time per run, but still in the same order of magnitude. However, since simulation is performed only after the collection of 32 new and different patterns, at least 32 SAT formulation and SAT solving runs take place for each simulation run.

The second observation is that, with few exceptions, the SAT formulation times are higher than the SAT solving times, which means that the SAT formulation procedure is successful in formulating SAT instances that are efficiently solvable. This was also the case for the application of TIGUAN to ISCAS and ITC'99 circuits (Table 4), and other authors have observed a similar relation between these times in their implementations [244]. Two notable exceptions are the circuits p77k and p469k. In these two cases, the SAT formulation time is not significantly above the average of all circuits, but the corresponding SAT solving times are several orders of magnitude higher than in all other circuits. This corroborates the observation that these two circuits tend to force testing tools to their worst-case behaviour. The average fault simulation times of these two circuits are also above the average, which suggests that the internal structure of these two circuits (which contain very long

[20] For better legibility, circuit p469k is not included in Figure 18.

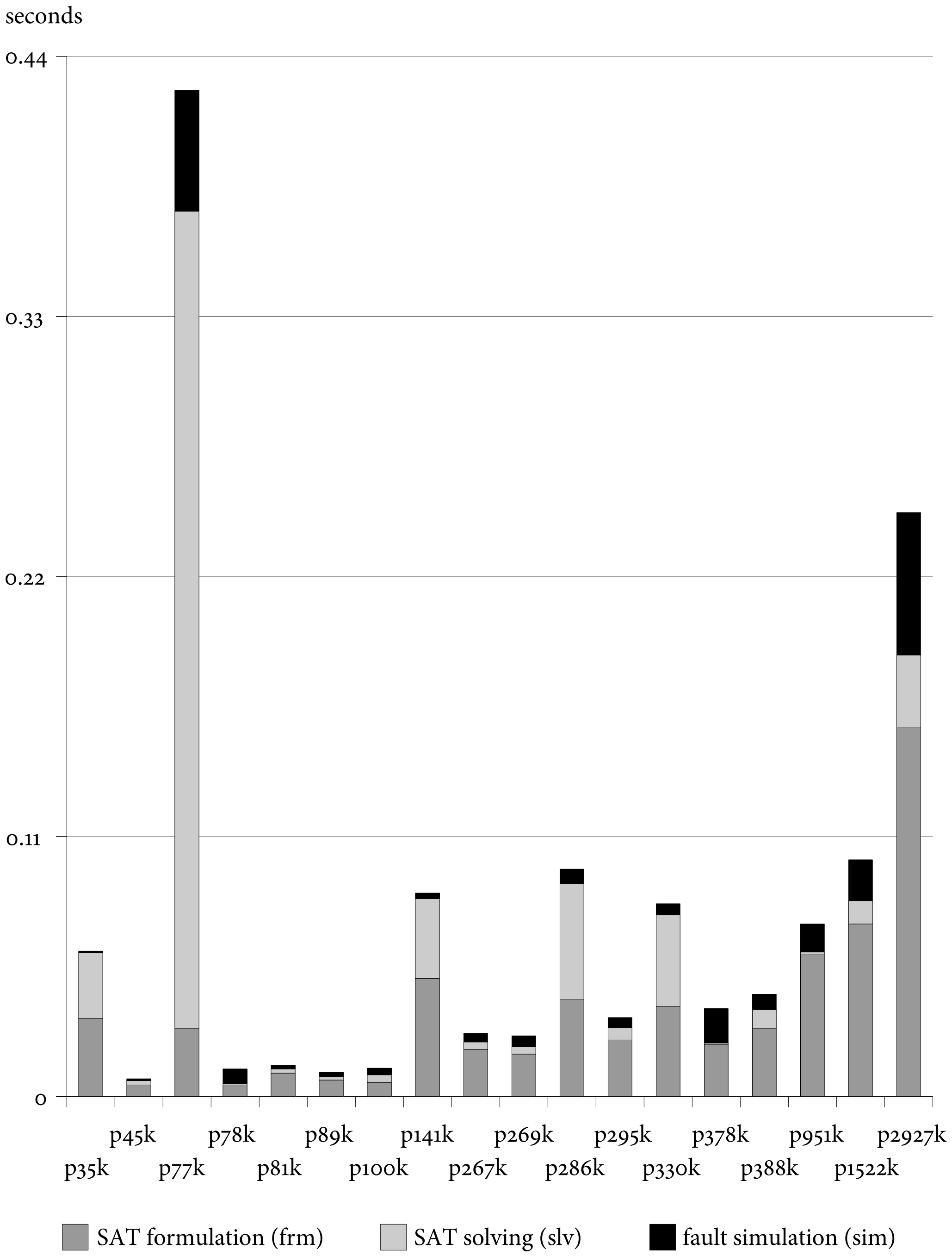


Figure 18. SAT-ATPG with 32-bit fault dropping for stuck-at faults — average times per run in comparison (columns 7–9 in Table 5)

paths) requires the traversing of very large parts of the search space to determine the detectability of a fault.

Table 6 compares Tiguan's performance (number of aborted faults — there were no aborts for itc'99 circuits — and total run-time in seconds) to the best published PASSAT results that were available at the time when these experiments were carried out [67]. The table includes only the circuits for which results on PASSAT's performance were available. The same parameters as in [67] were chosen, i.e. a timeout of 20 seconds per fault and the use of fault of dropping. For reference, a dual-core Xeon computer with 3 GHz and 32 GB RAM was used in [67]. Tiguan's numbers of aborted faults correspond to those shown in the fifth column of Table 5, while the run-times correspond to the last column of Table 5. PASSAT's results correspond to the fourth and fifth columns of Table VI in [67] (run-times were converted into seconds).

Except for circuits b15c and and p469k, Tiguan classifies more faults than PASSAT and has better run-times. A possible explanation for the notable difference observed for circuit p469k may be the fact that the results in [67] were obtained using MiniSat v1.14, whereas Tiguan uses MiraXT. Since different SAT solvers usually employ different sets of default parameters to guide their decision heuristics, they may perform differently when applied to certain types of SAT formulae despite having the same performance on average random SAT formulae. However, the large number of aborts produced by PASSAT for p469k in comparison to the other circuits, confirms the observation that p469k is a circuit that leads to particularly hard ATPG instances.

Circuits p89k, p141k and p951k contain tri-state elements[21]. Since Tiguan and its fault simulation back-end do not support multi-valued logic, bufif1 gates were replaced by and gates and invif1 gates were replaced by nand gates, which retains the circuit's functionality[22]. Hence, for these circuits, Tiguan's run-time advantage is partially owed to the simplified encoding of tri-state elements. PASSAT uses a four-valued logic that includes the high-impedance value z, but only in the circuit areas that can be influenced by tri-state elements, while the rest of the circuit is modelled using Boolean logic. According to [67], these three circuits contain 6.73%,

[21] *Tri-state elements* are buffers (bufif1) or inverters (invif1) with an additional control input. If the control input is set to logic 1, the gate behaves like a normal buffer or inverter. If the control input is set to logic 0, the gate's output adopts a high-impedance state that disconnects it from the circuit. Tri-state buffers are used to allow the output of several gates to be connected to one single line (a *bus*). The control signals of all tri-state elements driving one single bus are then set such that only one driver is active at a time.

[22] To prevent bus contention, additional clauses that ensure that at most one driver is active at the same time need to be generated.

Table 6

SAT-ATPG with 32-bit fault dropping for stuck-at faults — performance comparison with PASSAT [67]

	Tiguan	PASSAT
circuit	time (s)	time (s)
b14c	13.2	19.0
b15c	44.0	24.0
b17c	123.6	142.0
b18c	341.8	1,350.0
b20c	29.4	56.0
b21c	33.3	59.0
b22c	36.0	95.0

	Tiguan		PASSAT	
circuit	aborted	time (s)	aborted	time (s)
p35k	–	1,364.0	–	1,561.0
p81k	–	162.0	–	583.0
p89k	–	154.0	–	573.0
p100k	–	91.0	–	410.0
p141k	–	1,706.0	–	4,740.0
p469k	120	13,139.0	77	6,180.0
p951k	–	2,668.0	1	18,300.0

Table 7

SAT-ATPG without fault dropping for stuck-at faults — classification and performance comparison with PASSAT [86]

		classification		Tiguan		PASSAT	
circuit	faults	detected	undetectable	aborted	time (s)	aborted	time (s)
c0432	524	520	4	–	0.5	–	2.6
c0499	758	750	8	–	1.0	–	21.0
c1355	1,574	1,566	8	–	4.5	–	32.5
c1908	1,879	1,870	9	–	4.6	–	14.4
c3540	3,428	3,291	137	–	14.0	–	47.9
c7552	7,550	7,419	131	–	19.4	–	106.5
cs01494	1,506	1,494	12	–	0.6	–	2.7
cs05378	4,603	4,563	40	–	4.1	–	14.3
cs15850	11,725	11,336	389	–	47.8	–	121.3
cs38417	31,180	31,015	165	–	89.7	–	191.3
b10c	486	486	0	–	0.1	–	0.3
b11c	1,436	1,434	2	–	1.0	–	4.8
b12c	2,827	2,826	1	–	1.5	–	5.6
b14c	16,167	16,137	30	–	122.1	–	1,426.8
b15c	21,282	20,545	737	–	378.8	–	2,673.6
p81k	204,174	202,981	1,193	–	4,429	–	12,116
p89k	150,538	148,604	1,934	–	2,544	–	5,755
p100k	162,129	161,404	725	–	2,102	19	15,397
p141k	282,428	279,189	3,239	–	29,938	236	95,452
p951k	1,557,914	1,542,633	15,281	–	158,875	132	166,791

26.13% and 8.11%, respectively, of elements that required four-valued modelling. All other circuits are purely Boolean, and PASSAT uses only two-valued logic to model them.

In order to compare TIGUAN's and PASSAT's test generation algorithms independently of fault simulation, also experiments without fault dropping were performed. The results are shown in Table 7. The results quoted for PASSAT are the best numbers achieved by PASSAT among different learning techniques presented in [86]. For reference, an AMD Athlon computer with 2.2 GHz and 1 GB RAM was used in [86]. TIGUAN's results were generated with a timeout of 20 seconds per fault (as in [86]). Again, TIGUAN outperforms PASSAT regarding both aborts and run-time for all circuits.

Finally, the performance of TIGUAN was compared to a commercial ATPG tool that employs a structural algorithm. The results are listed in Table 8. The results are organised in five groups of columns, each consisting of a column *abr* which shows the number of aborted faults, and a column *time* which lists the total run-time in seconds. The first group corresponds to the application of TIGUAN to the combinational cores of larger ITC'99 circuits and to NXP circuits using a low timeout of 0.5 seconds per fault. The second group summarises TIGUAN's performance using a timeout of 20 seconds per fault. Note that these numbers are not the same as those quoted in Table 5, as these experiments were carried out on a different machine (a 2.3 GHz AMD Opteron computer with 4 GB RAM). The third group corresponds to TIGUAN's performance without a time limit. Hence, all faults were classified and this group contains no *abr* column. The experiment with a timeout of 20 seconds was started only for circuits for which TIGUAN was not able to classify all faults within the time limit of 0.5 seconds. Analogously, only circuits with aborted faults using a timeout of 20 seconds were considered for the experiment without a timeout. As expected, the run-times of the experiment with 20 seconds timeout are in general higher than those with a 0.5 seconds timeout, and the number of aborted faults is lower. An interesting finding is that there are circuits (b18c and p295k) for which the 20s-experiment has a lower run-time than the 0.5s-experiment even though more faults are classified. This stems from natural run-time variations that can be observed for the SAT solving of formulae for which the time needed to set up the internal data structures of the SAT solver is higher than the time needed for the search, which in general applies to formulae derived for SAT-ATPG, as will be discussed in Chapter 5. Since the initialisation of data structures involves mostly operations like memory allocation, its run-time can vary depending on the current workload of the operating system.

The run-time comparison between the 20s-experiment and the experiment without

Table 8

SAT-ATPG with 32-bit fault dropping for stuck-at faults — comparison with a commercial tool (structural ATPG)

	Tiguan					structural ATPG			
	0.5s timeout		20s timeout		no timeout	default		high conflict limit	
circuit	abr	time (s)	abr	time (s)	time (s)	abr	time (s)	abr	time (s)
b14c	–	14	–	–	–	3	4	2	1,708
b15c	–	46	–	–	–	166	63	1	10,437
b17c	–	122	–	–	–	481	171	4	18,920
b18c	1	395	–	392	–	247	250	56	121,274
b20c	–	32	–	–	–	6	15	4	2,393
b21c	–	38	–	–	–	7	10	6	3,407
b22c	–	41	–	–	–	2	16	2	4,173
p35k	–	1,529	–	–	–	2	74	1	6,936
p45k	–	54	–	–	–	15	14	–	1,053
p77k	2,554	4,311	–	5,346	–	381	1,475	unknown	> 700,000
p78k	–	7	–	–	–	–	10	–	–
p81k	–	153	–	–	–	201	69	65	51,356
p89k	–	198	–	–	–	2	33	–	356
p100k	–	103	–	–	–	260	90	11	22,928
p141k	1	1,847	–	1,889	–	24	144	13	18,468
p267k	–	474	–	–	–	–	119	–	–
p269k	–	506	–	–	–	17	121	–	12,828
p286k	129	2,099	1	3,260	3,497	78	313	11	13,637
p295k	5	1,229	1	1,164	1,228	19	307	12	7,013
p330k	93	2,781	31	3,272	23,475	186	426	36	26,827
p378k	–	52	–	–	–	–	62	–	–
p388k	31	879	2	934	1,263	66	272	7	23,903
p469k	9,170	13,611	147	14,695	30,815	210	4,189	unknown	> 700,000
p951k	–	2,781	–	–	–	193	1,692	–	14,595
p1522k	121	13,160	–	13,400	–	1,159	3,201	448	313,969
p2927k	1,189	34,475	92	39,724	50,812	5,601	4,108	165	264,202

time limit illustrates that the hard-to-solve ATPG instances of a circuit can be disproportionately harder than the same circuit's remainder of instances. For example, in the case of circuit p330k, 20,203 seconds are required to process the 31 faults that cannot be classified using a timeout of 20 seconds per fault, which means that these 31 faults require an average SAT solving time of more than 600 seconds, while this circuit has 540,665 faults with an average SAT solving time of less than 0.5 seconds.

The second-to-last group of columns in Table 8 corresponds to the performance of the commercial tool using its default parameters. In comparison to Tiguan, the commercial tool has lower run-times, but it aborts the processing of a considerable number of faults. The largest differences between TIGUAN's run-time and the commercial tool's run-time can be observed for the large NXP circuits, as these circuits have a very large number of faults, most of which are easy to detect. Hence, the structural ATPG algorithm has a better performance. However, the difference between TIGUAN's run-time and the commercial tool's run-time is less pronounced for the smaller ITC'99 circuits. For example, for circuit b17c, which has a large fraction of undetectable faults, TIGUAN is able to classify all faults in less time than the commercial tool, whereas the commercial tool even produces 481 aborts.

Finally, the last group of columns summarises the performance of the commercial tool using a very high conflict limit. Instead of using timeouts, the commercial tool aborts the processing of a fault when the search has reached a certain number of conflicts. Using this strategy allows the commercial tool to produce deterministic results regarding which faults are aborted. The experiment with the high conflict limit was applied only to circuits for which the tool was not able to classify all faults using default parameters. For circuits p77k and p469k, the experiment with the high conflict limit was aborted manually after 700,000 seconds. In contrast to TIGUAN, increasing the conflict limit leads to a disproportionate grow of the total run-time, which becomes higher than TIGUAN's highest run-time for all circuits, while there are still many faults that cannot be classified. Assuming that the faults aborted by the commercial tool using its default parameters constitute the group of faults that can be regarded as generally hard for structural ATPG algorithms, this shows that there is a subset of these faults which are particularly harder for structural than for SAT-based algorithms. This is especially evident for circuits p77k and p469k, for which the commercial tool using default parameters has a considerably better performance than TIGUAN. When the conflict limit was increased, the commercial tool was not able to classify all faults of these two circuits even after 700,000 seconds.

Table 9 shows the results of the last experiment, which was performed in order to evaluate the performance of the SAT-based algorithm for faults that are regarded as hard by the structural tool. The table shows the average run-time per fault that TIGUAN needed to generate the SAT formula (column group *SAT formulation*), to solve the SAT instance (column group *SAT solving*) and to perform the complete processing of a fault (column group *test generation*), for the set of all stuck-at faults (columns *all*) and for the set of faults that were aborted by the commercial tool using its default parameters (columns *hard*). The test generation time includes the SAT formulation time, the SAT solving time and also the time required for miscellaneous tasks like fault simulation.

The last line shows the average over all circuits. In general, the formulation of

Table 9

SAT-ATPG for stuck-at faults — evaluation of hard-to-detect faults

	average run-time per fault (s)					
	SAT formulation		SAT solving		test generation	
circuit	all	hard	all	hard	all	hard
p35k	0.0330	0.0380	0.0278	0.0220	0.0615	0.0600
p45k	0.0050	0.0040	0.0017	0.0029	0.0075	0.0069
p77k	0.0290	0.0350	0.3455	1.7769	0.4255	1.8124
p81k	0.0100	0.0090	0.0017	0.0019	0.0132	0.0109
p89k	0.0070	0.0240	0.0015	0.0600	0.0103	0.0840
p100k	0.0060	0.0080	0.0032	0.0352	0.0120	0.0432
p141k	0.0500	0.0450	0.0337	0.0247	0.0861	0.0697
p269k	0.0180	0.0260	0.0031	0.0092	0.0257	0.0352
p286k	0.0410	0.0320	0.0490	0.2941	0.0962	0.3274
p295k	0.0240	0.0300	0.0053	1.0714	0.0335	1.1014
p330k	0.0380	0.0220	0.0388	4.4320	0.0816	4.4564
p388k	0.0290	0.0390	0.0078	0.6604	0.0433	0.7017
p469k	0.0940	0.1060	4.4455	7.4815	6.2633	7.5889
p951k	0.0600	0.0590	0.0011	0.0090	0.0730	0.0745
p1522k	0.0730	0.1130	0.0099	0.0811	0.1002	0.2017
p2927k	0.1560	0.2050	0.0308	0.6693	0.2470	0.8877
average	0.0421	0.0496	0.3129	1.0395	0.4737	1.0914

the SAT instances for hard faults needs more time, but the average difference is of only 0.0075 seconds per fault, which shows that the SAT generation is largely unaffected by the fault hardness. In contrast, the average SAT solving time increases by a factor of 3.32, which shows that all faults regarded as hard by the structural algorithm are on average harder for the SAT-based algorithm as well, but the factor of 3.32 can be regarded as moderate, especially in comparison to the performance of the commercial tool with a high conflict limit. For circuit p469k, which is a hard benchmark, the increase factor is even smaller (1.68). The average increase factor of the total test generation time per fault is of only 2.30, which proves that SAT-ATPG performs particularly well on SAT instances that are too hard for structural algorithms.

4.6.2 CMS@ faults with non-empty aggressor sets

In order to evaluate Tiguan's performance on more complex faults, Tiguan was used to generate gate-exhaustive test sets. Table 10 summarises the application of

Table 10

CMS@-based SAT-ATPG with 32-bit fault dropping for gate-exhaustive testing — ISCAS'85, ISCAS'89 and ITC'99 circuits

		classification			run-time (s)	
circuit	faults	detected	undetectable	patterns	avg/flt	total
c1355	2,466	1,996	470	299	0.0009	2
c1908	5,440	3,846	1,594	552	0.0007	4
c2670	5,308	4,413	895	545	0.0006	3
c3540	10,358	5,364	4,994	596	0.0022	23
c5315	12,084	10,194	1,890	1,069	0.0004	4
c6288	9,664	7,934	1,730	439	0.0019	18
c7552	15,050	12,345	2,705	1,227	0.0006	9
cs01196	2,298	2,106	192	350	0.0002	0
cs01238	2,392	2,087	305	379	0.0003	1
cs01423	2,564	2,313	251	278	0.0002	1
cs01488	3,306	3,046	260	303	0.0001	0
cs01494	3,330	3,040	290	303	0.0001	0
cs05378	9,958	8,563	1,395	549	0.0002	2
cs09234	17,642	14,866	2,776	1,074	0.0005	9
cs13207	26,004	22,950	3,054	1,381	0.0006	15
cs15850	29,922	26,703	3,219	1,213	0.0009	26
cs35932	60,064	46,484	13,580	128	0.0004	23
cs38417	70,236	66,228	4,008	2,425	0.0003	20
cs38584	75,278	64,629	10,649	1,549	0.0003	24
b13c	1,246	1,132	114	98	0.0000	0
b14c	30,138	23,366	6,772	3,511	0.0023	70
b15c	41,402	29,718	11,684	2,144	0.0045	186
b17c	138,230	97,826	40,404	6,041	0.0040	554
b18c	396,886	292,165	104,721	16,084	0.0058	2,313
b20c	66,444	52,049	14,395	5,048	0.0028	187
b21c	66,420	52,444	13,976	5,597	0.0029	192
b22c	94,022	73,540	20,482	5,522	0.0026	244

TIGUAN to generate gate-exhaustive test sets for larger ISCAS and ITC'99 circuits, and Table 11 shows the results obtained for NXP circuits. The test generation was combined with 32-bit fault dropping (with random filling), and a timeout of 20 seconds per fault was imposed on SAT solving. No multi-threading was used. Note that the fault simulator was extended such as to respect the definition of the CMS@FM. That means, that a CMS@ fault is regarded as detected by simulation only if the simulated test pattern sets all aggressors to their corresponding values in addition to propagating a fault effect to a primary output.

Table 11

CMS@-based SAT-ATPG with 32-bit fault dropping for gate-exhaustive testing — NXP circuits

		classification				run-time (s)	
circuit	faults	detected	undetectable	aborted	patterns	avg/flt	total
p35k	150,898	136,980	13,918	–	19,049	0.0242	3,657
p45k	148,232	131,604	16,628	–	4,163	0.0009	140
p77k	254,176	215,505	38,667	4	12,271	0.0336	8,543
p78k	274,200	238,385	35,815	–	765	0.0009	258
p81k	404,348	361,938	42,410	–	28,773	0.0022	899
p89k	306,680	276,373	30,307	–	16,859	0.0016	496
p100k	328,516	293,707	34,809	–	7,195	0.0015	479
p141k	577,922	523,183	54,739	–	12,801	0.0102	5,871
p267k	850,024	781,926	68,098	–	18,631	0.0032	2,731
p269k	854,364	786,266	68,098	–	18,646	0.0032	2,717
p286k	1,265,588	1,091,567	174,020	1	28,619	0.0089	11,202
p295k	986,740	880,130	106,607	3	29,613	0.0054	5,351
p330k	1,166,046	1,037,130	128,843	73	36,401	0.0102	11,934
p378k	1,370,984	1,191,909	179,075	–	1,980	0.0037	5,117
p388k	1,663,442	1,463,686	199,754	2	17,317	0.0049	8,220
p469k	312,784	241,562	70,844	378	652	0.1618	50,603
p951k	3,250,198	2,884,773	365,425	–	28,050	0.0089	28,863
p1522k	3,708,692	3,350,769	357,923	–	80,404	0.0140	52,036
p2927k	7,048,378	6,253,392	794,723	263	51,340	0.0241	169,859

The second column of these two tables shows the number of targeted faults. The columns labelled *detected*, *undetectable* and *aborted* indicate how many faults were detected by pattern generation or by simulation, how many faults were proved undetectable, and how many faults were left unclassified due to a SAT solving timeout, respectively. The circuits on Table 10 had no aborted faults; hence, a column *aborted* is not included in that table. The next column (*patterns*) shows the amount of generated test patterns, which is significantly smaller than the number of detected faults due to the use of fault dropping, but no compaction techniques were employed. The last two columns quote the average run-time needed to process each fault (this time comprises SAT formulation, SAT solving and fault simulation), and the total run-time needed to process the whole fault list.

The total run-times are higher than the run-times measured for stuck-at faults, but this is due to the very large number of targeted faults, while the average time needed to process each fault is in the same order of magnitude. Thus, although these faults have non-empty aggressor sets, they are on average not significantly harder than stuck-at faults, which can be attributed to the presence of only one victim and to the fact that all aggressors of a CMS@ fault for gate-exhaustive testing are connected to the inputs of one single gate, which means that the area of the modelled part of the circuit is not increased in comparison to stuck-at faults. However, the larger number of aborted faults leads to the conclusion that hard-to-detect CMS@ faults with non-empty aggressor lists are on average harder than hard-to-detect stuck-at faults. Nonetheless, Tiguan was able to efficiently classify CMS@ faults with non-trivial additional conditions with only very few aborts.

In order to evaluate Tiguan's performance on even more complex faults, and in particular, in order to evaluate the impact the number of victims and aggressors has on a fault's hardness, Tiguan was used to generate tests for three sets of 10,000 randomly generated CMS@ faults. In contrast to CMS@ faults for gate-exhaustive testing, the aggressors and victims belonging to each random fault were allowed to be located anywhere on the circuit. Thus, with growing number of aggressors and victims, the faults are expected to be harder to process due to an increased area of the modelled part of the circuit. The experiment was applied with a timeout of 20 seconds per fault, and fault dropping was performed with a simulation width of 32. No multi-threading was used. Table 12 (a) lists the results. Columns 2–4 show the results for the first set of faults, where all faults had one aggressor and three victims. Columns 5–7 list the results for faults with five aggressors and five victims, and the last three columns report results on faults with ten aggressors and ten victims. The columns labelled *det* list the number of detected faults in each fault list, the columns labelled *und* list the number of provably undetectable faults, and the columns labelled *time* show the total run-time in seconds. The number of aborted faults is shown separately in Table 12 (b). Only the circuits p77k and p469k produced aborts, which repeatedly confirms that these circuits are particularly hard benchmarks.

The first observation that can be made is that the fraction of undetectable faults among the 10,000 faults and the total run-time both increase with growing number of aggressors and victims. This confirms the correlation between the undetectability of faults and the hardness of their corresponding ATPG instances (with respect to structural or SAT-based approaches), which has been previously observed for stuck-at faults. Although the fraction of undetectable faults grows also in the case of the industrial NXP benchmarks, it is lower among these circuits than among the smaller ISCAS and ITC'99 circuits, and this is especially evident for the CMS@ faults with ten aggressors and ten victims. This can be attributed to the depth of the NXP circuits, which is rather small in relation to their number of inputs and outputs, which means that the input cones of the primary outputs of these circuits

Table 12

SAT-ATPG with 32-bit fault dropping for sets of 10,000 random CMS@ faults

(a) classification and run-time

	1 aggressor, 3 victims			5 aggressors, 5 victims			10 aggressors, 10 victims		
circuit	det	und	time (s)	det	und	time (s)	det	und	time (s)
c1355	8,186	1,814	31.99	6,287	3,713	33.49	2,637	7,363	33.29
c1908	8,101	1,899	41.33	6,477	3,523	44.84	2,587	7,413	47.53
c2670	9,276	724	34.23	8,256	1,744	40.24	5,291	4,709	46.71
c3540	7,762	2,238	66.69	4,911	5,089	75.12	1,234	8,766	84.09
c5315	9,392	608	49.3	9,061	939	64.45	7,031	2,969	87.51
c6288	8,835	1,165	156.76	8,983	1,017	182.13	7,968	2,032	287.24
c7552	9,214	786	76.71	8,750	1,250	103.05	6,231	3,769	142.93
cs05378	9,220	780	27.47	8,424	1,576	40.48	5,801	4,199	64.78
cs09234	8,844	1,156	63.96	8,218	1,782	86.16	5,321	4,679	137.12
cs13207	9,293	707	38.73	8,940	1,060	61.26	6,961	3,039	88.94
cs15850	9,156	844	82.8	8,876	1,124	116.96	7,034	2,966	180.26
cs35932	9,204	796	33.01	8,570	1,430	47.71	6,357	3,643	65.02
cs38417	9,429	571	62.43	9,736	264	89.91	9,325	675	171.95
cs38584	8,825	1,175	51.06	7,504	2,496	64.36	4,724	5,276	102.17
b13c	9,530	470	4.5	7,689	2,311	6.13	3,592	6,408	9.45
b14c	8,818	1,182	199.32	7,362	2,638	261.55	4,584	5,416	336
b15c	8,150	1,850	352.59	5,254	4,746	398.55	1,817	8,183	489.5
b17c	9,306	694	473.15	8,238	1,762	656.99	5,167	4,833	1,044.97
b18c	9,576	424	719	8,829	1,171	1,061.83	6,514	3,486	1,710.64
b20c	9,277	723	352.13	8,767	1,233	523.53	6,758	3,242	749.46
b21c	9,353	647	353.66	8,739	1,261	547.62	6,726	3,274	731.52
b22c	9,457	543	377.55	9,196	804	618.42	7,593	2,407	881.88
p35k	9,297	703	1,413.24	9,400	600	1,581.93	8,623	1,377	1,551.26
p45k	9,388	612	145.49	9,104	896	222.83	7,539	2,461	356.12
p77k	8,846	1,138	5,254.23	8,049	1,943	3,479.69	5,708	4,292	2,447.43
p78k	9,556	444	163.3	9,760	240	255.92	9,441	559	468.65
p81k	9,546	454	366.8	9,778	222	591.78	9,582	418	1,275.26
p89k	9,552	448	246.69	9,720	280	386.52	9,221	779	773.97
p100k	9,511	489	205.97	9,674	326	315.16	9,029	971	699.18
p141k	9,465	535	1,653.21	9,633	367	2,604.32	9,307	693	4,056.55
p267k	9,434	566	550.17	9,806	194	617.73	9,605	395	1,083.35
p269k	9,419	581	512.65	9,798	202	726.02	9,582	418	1,068.83
p286k	9,410	590	931.46	9,533	467	1,133.44	8,653	1,347	1,891.96
p295k	9,391	609	618.43	9,036	964	687.67	7,280	2,720	1,018.1
p330k	9,420	579	921.99	9,819	181	1,571.12	9,545	455	2,263.24
p378k	9,567	433	521.98	9,796	204	518.48	9,419	581	742.96
p388k	9,497	503	621.94	9,911	89	836.72	9,895	105	1,344.34
p469k	6,623	1,867	79,985.77	4,808	1,541	114,489.46	1,701	1,394	157,864.57
p951k	9,560	440	927.9	9,786	214	1,229.66	9,530	470	1,181.92
p1522k	9,511	489	1,965.6	9,743	257	2,457.8	9,390	610	5,230.21
p2927k	9,196	804	3,359.97	9,455	545	2,479.48	8,958	1,042	4,043.21

(b) aborted faults

circuit	1 agg, 3 vic	5 agg, 5 vic	10 agg, 10 vic
p77k	16	8	–
p469k	1,510	3,651	6,905

are relatively narrow, and hence, that the probability that the conditions imposed on the ten random aggressors conflict with each other is lower than in the smaller circuits.

With the exception of p77k and p469k, Tiguan is able to classify all faults without aborts and using the same time limit that was used to classify stuck-at faults, which are intuitively less complex than the random CMS@ faults shown here, as single-stuck-at faults have only one victim and no aggressors. Furthermore, it can be observed that the increase of total run-time is less severe than the increase of undetectable faults, which shows that the application of Tiguan to such hard faults benefits from the advanced SAT-based learning techniques. Thus, Tiguan is especially suitable for complex fault models that allow the definition of multiple fault effects and the imposition of multiple conditions for fault activation.

4.7 Conclusions

In this chapter, the SAT-based test pattern generator Tiguan was presented. Tiguan can classify all single-stuck-at faults in large industrial circuits and in structurally complex iscas circuits, even without the utilisation of the SAT engine's multithreading capability. Tiguan outperforms the tool PASSAT developed at the University of Bremen, which constituted the state-of-the-art in SAT-ATPG at the time when the experiment was performed. Also a comparison to a commercial tool that uses a structural algorithm was presented. The analysis of the results proves that SAT-based ATPG performs particularly well for faults that are too hard for structural algorithms.

Tiguan works internally with the CMS@FM, a generic fault model that enables the representation of multiple victim lines with stuck-at behaviour as well as conditional fault activation in function of multiple aggressor lines. Experiments on CMS@ faults for gate-exhaustive testing and on random CMS@ faults with many aggressors and many victims show that Tiguan is especially suitable for the application to complex fault models.

An important result is that the time needed for SAT formulation is not only non-negligible, but on average even higher than the time needed for SAT solving. In the second part of the next chapter, a fault clustering technique is presented which eliminates a large number of SAT formulation runs, and reduces the SAT solving time thanks to incremental SAT solving, thus significantly increasing Tiguan's run-time efficiency.

5

Optimising the run-time of SAT-based ATPG

After the introduction of the SAT-ATPG tool Tiguan in the previous chapter, this chapter focuses on techniques to improve Tiguan's run-time efficiency. The first part of the chapter gives insight into the algorithms used by the SAT engine MiraXT[23]. This is done with the purpose of explaining some of the modifications made to MiraXT for a better performance on SAT formulae generated by Tiguan[24], and of introducing MiraXT's implementation of multi-threading, which is essential to understand the evaluation of Tiguan's performance on multi-core architectures that follows. The second part of the chapter introduces the SAT engine antom and explains the most important differences between antom and MiraXT from the point of view of a SAT-ATPG application. Then, a fault clustering technique that utilises antom's incremental SAT solving is presented and evaluated.

Author's contribution — The author's contribution consisted in the performance evaluation of SAT-ATPG on multi-core architectures, and the engineering of a two-stage method to best exploit MiraXT's multi-threading capability. In addition, the author integrated the new SAT engine antom into Tiguan, and developed the fault clustering technique, paying special attention to preserving the efficiency of existing code base.

Parts of the work covered in this chapter have been published in [J2, C16, C7, W10, W9, W7, W5, W2] (see author's publications on pages 223–226).

[23] See Section 3.3 for a general introduction to algorithms for the solution of the SAT problem.

[24] These modifications were part of the initial implementation process of Tiguan. Hence, the experimental results shown in Section 4.6 already reflect these optimisations.

5.1 SAT-ATPG with thread-parallel SAT solving

As processor design has approached the limits of speed scalability, the last decade has brought about multi-core architectures as a way of further optimising the run-time of computationally complex tasks. Processes can distribute tasks among several computation threads, and all threads can run in parallel, each on an own processor or processor core, thus reducing the real time needed to complete the task. However, as different threads have to communicate with each other, and due to task dependencies between threads, the speed-up scaling is usually sub-linear in the number of threads.

Several algorithms in the fields of electronic design automation and testing have been ported to multi-core architectures such as to benefit from this new paradigm of computing. The SAT-based test pattern generator Tiguan, which supports multi-threaded SAT solving, is an example of such a tool. The relevant question is how the tool can benefit from these architectures in an optimal way. Significant research on distributed ATPG has been performed in the past [88], albeit based on the general assumption that the communication between processor nodes is very expensive. However, modern multi-core architectures have largely rendered this assumption invalid thanks to design principles that locate individual cores in physical proximity of each other. Also fast buses (such as AMD's HyperTransportBus) and multi-level cache design that enables fast communication between threads have increased the usability of multi-core architectures.

In this section, Tiguan's performance on multi-core architectures is evaluated experimentally using the SAT solver MiraXT, which is able to distribute the SAT solving process among several computation threads. In this context, it is important to consider the average properties of SAT instances derived from ATPG in comparison to the type of SAT instances that have driven the development of modern SAT solvers, namely instances derived from problems in the field of formal verification. Here, the typical workload is characterised by few, but very hard and usually unsatisfiable SAT instances (e.g. equivalence checks or property checks). This means that most modern SAT solvers have been tuned for efficiency on instances whose hardness can be regarded as above-average from the point of view of ATPG. In fact, it was shown in the previous chapter that more than 90% of the SAT instances derived from ATPG for large industrial circuits can be solved very fast, even without the use of sophisticated speed-up techniques. This poses the question whether the optimisation techniques found in modern SAT solvers are suitable for the solution of the average SAT-ATPG instance. In order to answer this question, this section analyses the performance of the various stages of the SAT solving process.

In addition, it was shown in Section 4.6 that the few hard SAT-ATPG instances can be disproportionately harder than the rest. In [67], the same observation motivated a two-stage approach, where a commercial structural ATPG tool was first applied to all faults, while a SAT-based ATPG tool (without thread parallelism) was applied only to the faults aborted by the structural ATPG engine. In this section, this principle is applied to a purely SAT-based solution, exploiting the fact that TIGUAN is able to dynamically control the internal parameters of the SAT solver MiraXT.

5.1.1 THE SAT SOLVING BACK-END MIRAXT

This section briefly introduces the SAT solver's architecture and some of the algorithmic details that were modified to optimise its performance as a SAT solving back-end for TIGUAN. More information on the SAT solver's internals can be found in [151].

Given a SAT formula φ in CNF, MiraXT computes a model of φ if φ is satisfiable. As a pre-processing step, MiraXT applies the unit propagation rule — the triggering clauses of a SAT formula produced by TIGUAN may include several unit clauses — and the pure-literal rule. MiraXT is also able to apply further pre-processing techniques. Along with variable elimination through resolution and clause elimination through subsumption, MiraXT uses a *unit propagation look-ahead* routine (UPLA) similar to techniques employed by Stålmarck's Proof Procedure [223] and to techniques implemented in *March_eq* [115] and *RSAT* [178]. This technique consists in assigning 1 to any yet unassigned variable X_1 and recording all implications of that assignment, then assigning the variable to 0 and recording also that assignment's implications, and finally analysing both sets of implications according to a fixed set of rules. For example, if X_1 implies X_2 and $\neg X_1$ implies $\neg X_2$ for a variable X_2, then both variables are equivalent and all occurrences of X_2 can be replaced by X_1.

However, these three advanced techniques did not prove beneficial for TIGUAN. The time needed to perform the required analysis exceeded the speed-up that these techniques produced on the later solving. Hence, these pre-processing steps were permanently switched off for MiraXT's use in TIGUAN.

One reason why TIGUAN did not benefit from MiraXT's advanced pre-processing is the specific structure of SAT formulae derived from ATPG. First of all, due to TIGUAN's efficient assignment of Boolean variables, the generated SAT formulae do not contain the amount of redundancy that the subsumption technique intends to eliminate. Second, although SAT formulae derived from the ATPG problem can have very large numbers of clauses and variables, most variables appear in only

Table 13
Influence of VSIDS on the SAT solving time

	old VSIDS		new VSIDS	
circuit	avg slv (s)	total (s)	avg slv (s)	total (s)
b14c	0.0021	21	0.0017	18
b15c	0.0066	62	0.0052	55
b17c	0.0059	194	0.0052	186
b18c	0.0039	672	0.0031	576
b20c	0.0040	74	0.0032	67
b21c	0.0035	72	0.0030	67
b22c	0.0039	101	0.0032	83
norm avg	100%	100%	82.3%	74.4%

few clauses as all variables stand for circuit lines and the SAT formula contains only clauses that express those lines' relationship to other lines in their immediate vicinity. In addition, except for triggering clauses, which can have as many variables as there are coloured primary outputs, clause size is bound by the number of ports of the modelled gates. As a consequence, the application of UPLA was not able to reveal useful global implications.

Further preliminary experiments showed that also some of MiraXT's parameters which guide the search need to be modified for better performance in SAT-ATPG. For example, although the search procedure of modern SAT solvers is in principle DPLL-based, some solvers assign a random value to all Boolean variables at the beginning, and then try to resolve all conflicts rather than satisfy every single clause. This results in an equally valid solution, but the search is guided in a more intelligent way. For MiraXT's application in Tiguan, experimental data revealed that it is better to assign fixed values (for example, only 0 or only 1) instead of random values at the beginning. Also MiraXT's decision strategy was modified. MiraXT employs the VSIDS decision strategy (Variable-State Independent Decaying Sum, see Section 3.3.2). In VSIDS, every variable is assigned a counter that is incremented at every occurrence of the variable in a clause. Due to Tiguan and MiraXT's tight integration, the initial values of these counters are computed on the fly as soon as Tiguan submits a new clause. During the solving process, the counters only need to be incremented when new learnt clauses are added to the clause database. Then, whenever a variable needs to be chosen, the variable with the largest number of occurrences is chosen. In order to enable the identification of variables with the most occurrences, the variable list must be sorted after each update of a counter, which MiraXT implements using bucket sort.

After the addition of originally 512 new learnt clauses, all variable counters are divided by two (efficiently implemented with a right shift). This is called a decay operation and has the aim of giving variables that occur in newer conflict clauses preference over those in older conflict clauses. Experiments determined that increasing the frequency of decay operations allows MiraXT to solve TIGUAN's SAT instances in shorter time. Table 13 shows the average SAT solving times per fault and the total run-times for larger ITC'99 circuits, using MiraXT's default VSIDS strategy (column group *old VSIDS*), and using the strategy with an increased frequency of decay operations (column group *new VSIDS*). The last row quotes the normalised average. Increasing the frequency of decay operations lead to a decrease of total run-time of about 25%. The reason for this is that more frequent decay operations translate into new conflict clauses being given even more preference over older conflict clauses. Such a decision strategy is better suited to TIGUAN's SAT instances due to the rather limited scope of their variables.

Along with *early-conflict-detection*-based BCP and other innovations like *implication queue sorting*, one of MiraXT's most prominent features is that it was the first modern SAT solver that took advantage of modern multi-core processors [151]. Figure 19 depicts MiraXT's design architecture. The basis is a shared-memory system that runs within the bounds of one single process. Hence, the different computation threads, which can be executed in parallel on different processor cores, can share data structures, global variables and pointers, and are thus able to access shared information at any time. In particular, conflict clauses learnt by one thread are immediately available to all other threads.

The shared-memory architecture allows the threads to communicate with each other faster than in message-passing-based systems. However, shared memory requires the implementation of systems to cope with race conditions and data hazards. In order to avoid the reduction of concurrency, all of MiraXT's operations are implemented in a manner that requires only few locks. For instance, data structures that are frequently used, like watched literals and VSIDS counters, are either stored in each thread's private part of the memory and not shared, or they are shared in read-only form. In MiraXT, locks are needed for only three activities:

- adding new conflict clauses and deleting old ones,
- generating and receiving new sub-problems,
- reporting a final solution if one exists.

Adding and deleting conflict clauses is the most common reason for the need of locks. In order to reduce the number of required locks, each thread analyses clauses recently inserted by other threads before inserting a new clause. This analysis can

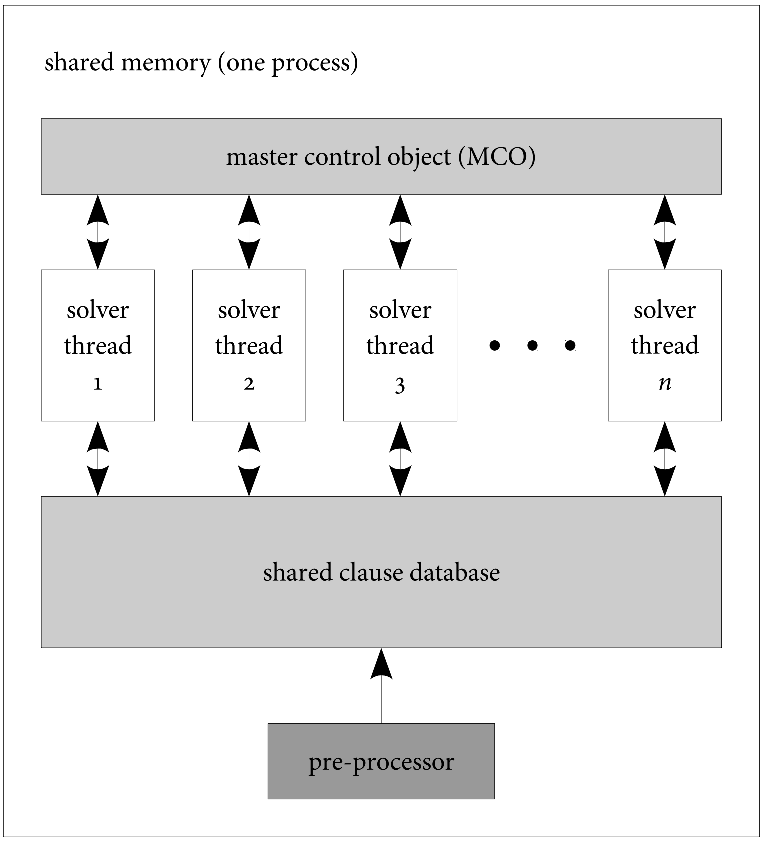


FIGURE 19. MIRAXT — DESIGN ARCHITECTURE

be done without locks as the other clauses are not modified by it. If they are found to be more effective, e.g. because they prune larger parts of the search space, the new clause is not inserted at all. In addition, the deletion of clauses is implemented by a two-stage garbage collection strategy that almost eliminates the need for locks.

After the application of the pre-processor, which operates directly on the clause database in a single-threaded fashion, the solving procedure is computed by several threads running in parallel, where each thread is in charge of a portion of the search tree. This approach is known as *search-space splitting* [153]. Running and idle threads are coordinated by a *master control object* (MCO) of very limited complexity which never intervenes in the threads' computation process.

At the beginning of the solving, the complete decision tree is given to one of the threads. All other threads communicate to the MCO that they are idle. Idle threads are put into sleep mode in which they do not poll and consequently do not cause communication overhead. Running threads poll the MCO periodically whether any global events have occurred. Example global events are "SAT instance has been solved", "timeout has been exceeded" or "a thread has become idle". In the latter case, the MCO makes one of the running threads divide its sub-problem into two parts, wakes one of the idle threads and transfers control of one problem part to that thread.

If a thread determines its sub-problem to be unsatisfiable, it inserts the derived conflict clauses into the database and enters the idle state. The process terminates when either one thread finds a satisfying assignment, or if the MCO eventually determines that all threads have become idle. In the latter case, the complete search space has been exhausted and the SAT instance is identified as unsatisfiable.

5.1.2 Performance of Tiguan on multi-core architectures

A series of experiments were performed on large industrial benchmarks provided by NXP. All experiments presented in this section were executed on a 2.3 GHz AMD Opteron computer with 4 Quad-Core processors (hence, up to sixteen computation threads can run in parallel, each on an own processor core) and 64 GB RAM. Note, however, that MiraXT is a 32-bit application; hence, the memory use of a Tiguan process is limited to a total of 4 GB independently of the number of SAT solving threads.

It has been noted before, e.g. in [108], that sophisticated performance enhancements are effective for relatively few hard-to-detect faults while the processing of easy-to-detect faults tends to be slowed down. The same could be observed for Tiguan in preliminary experiments, which is why some of MiraXT's pre-processing techniques were switched off permanently. The next question is whether the same applies to multi-threaded SAT solving. Table 14 shows two extreme examples: circuit p45k, a circuit with 99.7% detectable faults, for which the average SAT solving time per fault amounts to 0.0017 seconds and the largest measured SAT solving time lies below 0.5 seconds (see Table 5); and circuit p469k which has been proved to be a hard benchmark, with an average SAT solving time per fault of 4.4455 seconds and a maximal SAT solving time of several thousand seconds. In this experiment, all stuck-at faults were targeted using different numbers of computation threads between 1 and 16 and a fixed timeout of 3 seconds per fault. This time limit was chosen such as to be selective on p469k. The table shows the average run-time per fault of different stages of fault processing, the total run-time in seconds (columns *total*) and the number of aborted faults (column *abr*) for these two circuits (no faults were aborted for p45k). The columns labelled *thr* quote the *thread initialisation*

Table 14

SAT-ATPG with multi-threaded SAT solving for stuck-at faults

	p45k				p469k				
	run-time (s)					run-time (s)			
	average/fault					average/fault			
threads	thr	slv	tpg	total	abr	thr	slv	tpg	total
1	0.0039	0.0039	0.0165	83	917	0.1135	2.2389	2.4978	16,229
2	0.0079	0.0045	0.0200	99	284	0.1764	1.0810	1.3810	5,447
3	0.0107	0.0027	0.0204	111	123	0.2540	0.8553	1.2270	4,941
4	0.0133	0.0046	0.0246	123	53	0.3477	0.7454	1.2128	5,150
5	0.0152	0.0024	0.0240	139	27	0.4496	0.6885	1.2581	5,659
6	0.0184	0.0061	0.0307	154	18	0.5188	0.6409	1.2752	5,487
7	0.0219	0.0051	0.0336	177	10	0.6303	0.6572	1.4085	6,085
8	0.0257	0.0032	0.0356	197	1	0.6904	0.5806	1.3879	6,264
9	0.0266	0.0048	0.0375	206	0	0.8028	0.6645	1.5941	6,886
10	0.0294	0.0040	0.0399	221	1	0.8962	0.6751	1.6975	7,295
11	0.0335	0.0029	0.0429	243	3	0.9845	0.6765	1.7884	7,346
12	0.0364	0.0040	0.0469	259	5	1.0679	0.6909	1.8844	7,896
13	0.0417	0.0058	0.0540	289	1	1.1467	0.7070	1.9691	8,217
14	0.0447	0.0043	0.0555	302	8	1.2398	0.7509	2.1063	8,911
15	0.0491	0.0040	0.0597	326	1	1.3225	0.6233	2.0582	8,543
16	0.0509	0.0061	0.0637	353	2	1.4276	0.8399	2.3848	9,801

time, i.e. the time that the SAT solver needs to set up the data structures for the different computation threads prior to the start of the actual solving process. The columns labelled *slv* quote the run-time needed for the actual SAT solving process, i.e. for the search for a Boolean solution. Finally, the columns labelled *tpg* list the total run-time needed to process one fault. It includes the thread initialisation time, the SAT solving time and the time needed for other tasks that are not performed in thread-parallel fashion, like SAT formulation and fault simulation.

It can be seen that the thread initialisation time increases linearly with the number of threads, and the same was observed for other circuits that are not shown in the table. Thread initialisation includes tasks like allocating memory for pointers to the shared clause database and is independent of the hardness of the SAT instance. Instead, this run-time depends on the size of the SAT formula in terms of the number of contained clauses and variables and on the efficiency of the operating system. The results show that this time also depends linearly on the number of threads, i.e. on the amount of memory to allocate and data structures to initialise. For all these reasons, the thread initialisation time cannot be optimised. In contrast, the average SAT solving time varies depending on the number of computation threads. In the case of p45k, where all instances can be regarded as easy to solve, the communication overhead leads to variations that cannot be analysed systematically. In general, the average SAT solving time remains in the same order of magnitude for all numbers of threads. In combination with a linearly growing thread initialisation time, the average test generation time per fault increases consistently with a growing number of threads, and so does the total run-time. This suggests that the application of multi-threaded SAT solving is ineffective for circuits like p45k with only easy-to-solve ATPG instances.

A different trend can be observed for the particularly hard circuit p469k. Here, a higher number of threads leads not only to fewer aborted faults, but also reduces the average SAT solving time which reaches its best value for 8 computation threads. However, using more than 8 threads does not reduce the average SAT solving time any further, which, combined with a linearly growing thread initialisation time, leads to test generation times per fault that grow to nearly the same value as in the experiment with 1 thread. However, for large numbers of threads, the number of aborted faults remains low since the growing test generation time stems largely from the large thread initialisation times, while the average SAT solving time is still better than in the experiments with 1 or 2 threads, even as it does not improve for thread numbers greater than 8. Altogether, it can be seen that the quality of test generation for hard-to-detect faults benefits from multi-threading both in terms of number of classified faults and total run-time. However, the best results are observed for about 8 computation threads, while a larger number of threads does not lead to further improvement.

In order to achieve a better understanding of the impact of multi-threading on SAT-ATPG for hard-to-detect faults, a second experiment was performed considering only hard circuits and only faults that were aborted by a commercial structural ATPG tool using its default parameters (see also Table 8). No SAT solving timeout was imposed on Tiguan; hence, all considered faults were classified either as detectable or as provably undetectable. Table 15 reports the average times per fault and the total run-times (columns labelled *tot*). The meaning of the columns labelled *thr*, *slv* and *tpg* is the same as in Table 14. These data are also shown in graphical form in Figure 20.

Like in the previous experiment, it can be seen that the thread initialisation time increases linearly with the number of threads. However, since only hard-to-detect faults are considered and the average SAT solving times are higher than in the first experiment, the thread initialisation times lie below the average SAT solving times in all cases. Regarding the average SAT solving time, it decreases rapidly for up to

Table 15
SAT-ATPG with multi-threaded SAT solving for hard-to-detect stuck-at faults

	p295k (19 faults)				p330k (186 faults)				p388k (66 faults)			
	avg/flt (s)				avg/flt (s)				avg/flt (s)			
threads	thr	slv	tpg	tot (s)	thr	slv	tpg	tot (s)	thr	slv	tpg	tot (s)
1	0.01	3.61	3.65	72	0.00	83.77	83.79	15,588	0.01	4.38	4.42	295
2	0.03	2.63	2.69	73	0.00	58.84	58.87	10,965	0.01	3.35	3.39	251
3	0.04	1.68	1.75	55	0.01	31.45	31.48	5,880	0.02	2.94	2.99	227
4	0.05	1.16	1.24	47	0.01	26.20	26.24	4,906	0.02	2.79	2.86	217
5	0.06	1.95	2.04	51	0.01	17.94	17.97	3,395	0.03	2.38	2.44	189
6	0.07	1.53	1.64	55	0.01	13.78	13.82	2,596	0.03	1.80	1.87	154
7	0.09	0.79	0.91	40	0.02	13.48	13.52	2,546	0.04	1.91	1.98	160
8	0.13	1.11	1.27	47	0.02	12.49	12.53	2,356	0.04	2.20	2.28	181
9	0.11	1.32	1.46	53	0.02	10.63	10.67	1,997	0.05	2.14	2.22	177
10	0.12	0.89	1.05	45	0.03	10.38	10.43	1,972	0.06	2.02	2.11	173
11	0.13	2.16	2.33	67	0.03	8.21	8.25	1,567	0.06	1.73	1.82	148
12	0.15	1.16	1.34	48	0.03	7.69	7.75	1,470	0.06	1.80	1.90	160
13	0.18	1.68	1.90	59	0.03	7.77	7.83	1,485	0.07	1.74	1.85	154
14	0.17	1.05	1.26	43	0.04	7.34	7.40	1,412	0.07	1.79	1.89	158
15	0.19	0.89	1.12	45	0.04	6.34	6.40	1,221	0.08	1.80	1.92	160
16	0.21	1.21	1.46	50	0.04	6.23	6.29	1,204	0.08	1.73	1.85	153

	p469k (210 faults)				p2927k (5,601 faults)			
	avg/flt (s)				avg/flt (s)			
threads	thr	slv	tpg	tot (s)	thr	slv	tpg	tot (s)
1	0.08	14.12	14.32	3,026	0.03	4.12	4.29	24,204
2	0.19	5.14	5.46	1,186	0.06	2.64	2.90	16,566
3	0.25	2.67	3.03	686	0.08	2.01	2.28	13,207
4	0.33	2.09	2.54	586	0.11	1.58	1.86	10,970
5	0.43	1.93	2.48	597	0.14	1.49	1.82	10,873
6	0.49	1.48	2.08	510	0.16	1.24	1.56	9,498
7	0.60	1.85	2.57	636	0.18	1.21	1.55	9,459
8	0.69	1.70	2.51	643	0.22	1.28	1.70	10,452
9	0.74	1.57	2.43	614	0.24	1.13	1.53	9,601
10	0.87	1.71	2.70	699	0.27	1.28	1.73	10,853
11	0.91	1.62	2.65	688	0.28	1.08	1.52	9,682
12	1.00	1.83	2.95	765	0.31	1.12	1.59	10,174
13	1.13	1.91	3.16	829	0.35	1.24	1.76	11,227
14	1.17	1.64	2.93	787	0.36	1.17	1.70	10,955
15	1.21	1.73	3.05	807	0.38	1.17	1.71	11,041
16	1.32	1.76	3.19	864	0.41	1.16	1.73	11,269

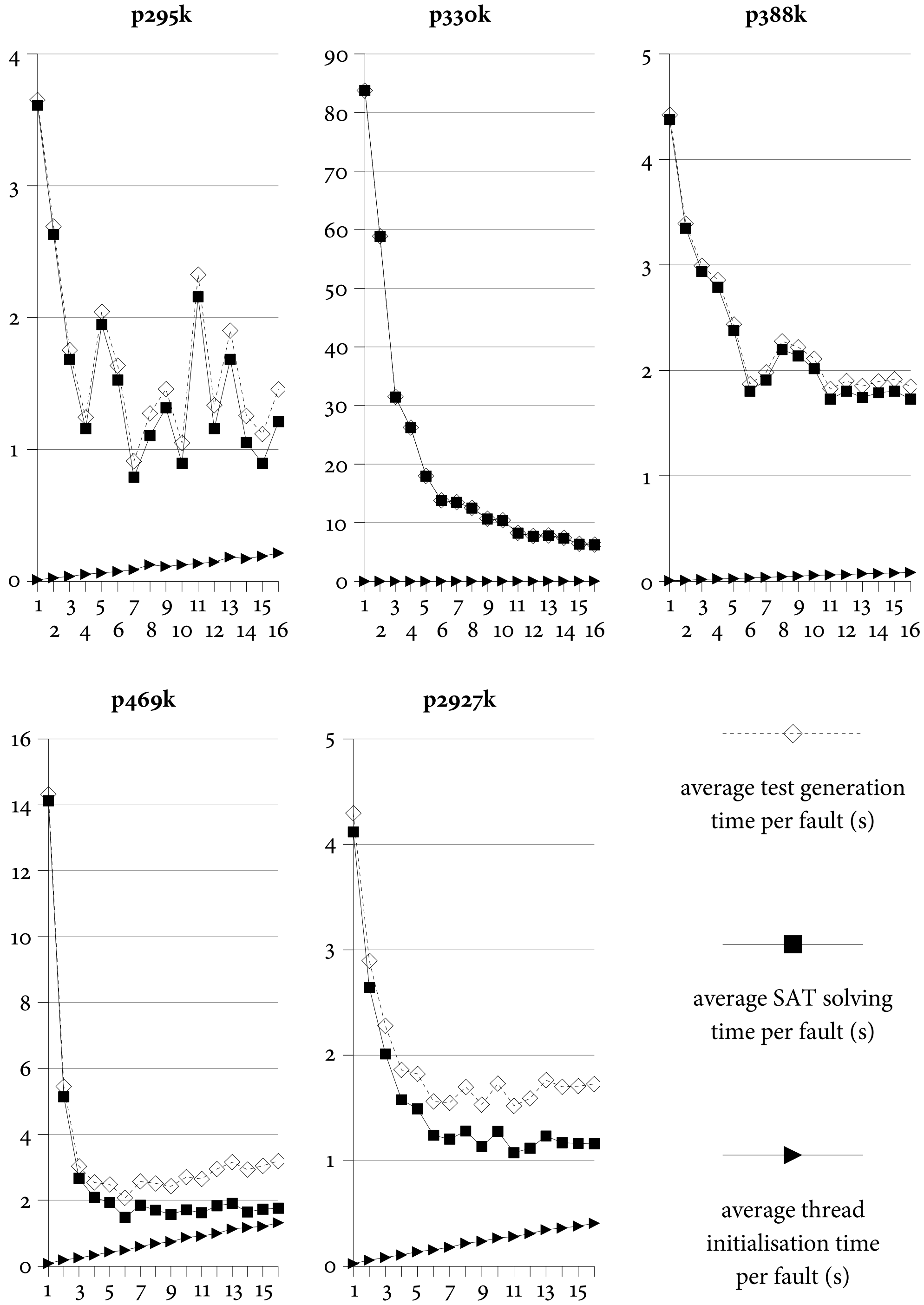


Figure 20. Data from Table 15 in graphical form

4–6 computation threads and then stagnates, fluctuates randomly or even increases. Hence, the behaviour of the average SAT solving time in function of the number of threads is the same behaviour that was observed for p469k in the first experiment. A possible reason for this phenomenon is the constant-rate learning performed by the individual threads. In this experiment, each thread generates roughly 10,000 new conflict clauses per second. Due to MiraXT's shared-memory architecture, this leads to memory congestion, which slows down the solving process when the number of threads is too high.

The overall observation is that SAT-ATPG for hard-to-detect faults benefits strongly from thread-parallel SAT solving up to a certain number of computation threads, while SAT-ATPG for easy-to-solve instances is consistently slowed down due to the large thread initialisation time. Consequently, a two-stage ATPG strategy was implemented. In the first stage, TIGUAN was run with an aggressive SAT solving time limit of 1 second per fault in order to filter out harder SAT instances. In the second stage, TIGUAN was applied to the remaining faults (hard-to-detect faults) with a higher timeout of 20 seconds per fault and employing thread parallelism. Table 16 (a) summarises the results for circuits with at least one abort during the first stage. The second and third columns quote the run-time of the first stage and the number of faults aborted during the first stage, respectively. This is also the number of faults targeted in the second stage. The remaining columns are organised in three groups reporting the performance of the second stage using multi-threaded SAT solving with 1, 2 and 4 threads. The columns labelled *abr* quote the number of faults that remained unclassified after the second stage, while the columns labelled *time* and *total* quote the total run-time in seconds of the second stage and the accumulated run-time of both stages, respectively.

Table 16 (b) compares the best results observed for the two-stage approach to the results obtained by the one-stage approach with a timeout of 20 seconds and without a timeout (there are no aborts in this case). The experiment without timeout was only performed for circuits where not all faults could be classified with a timeout of 20 seconds. The second column indicates the number of threads employed during the second stage that leads to the results quoted in the third and fourth columns.

It is apparent that the two-stage method is advantageous for several circuits. With a higher number of employed threads, the number of faults the second stage manages to classify is consistently better, and also the run-times of the second stage are better, especially for hard circuits like p469k. However, there are circuits for which the second stage yielded better results using a lower number of threads. This applies to circuits with either only few faults targeted in the second stage or with hard-to-detect faults that are not too hard on average. In comparison to the one-stage

Table 16
Two-stage thread-parallel SAT-ATPG for stuck-at faults

(a) two-stage approach

	first stage		second stage (20s timeout)								
	(1s timeout)		1 thread			2 threads			4 threads		
circuit	time (s)	abr	abr	time (s)	total (s)	abr	time (s)	total (s)	abr	time (s)	total (s)
p77k	4,545	1,322	–	2,940	7,485	–	1,354	5,899	–	1,003	5,548
p286k	2,115	126	1	1,459	3,574	1	1,232	3,347	1	1,609	3,724
p295k	1,062	3	1	45	1,107	1	62	1,124	1	66	1,128
p330k	2,376	70	31	806	3,182	17	616	2,992	16	491	2,867
p388k	800	2	2	40	840	2	41	841	2	40	840
p469k	17,929	2,680	141	10,434	28,363	28	3,343	21,272	3	2,152	20,081
p1522k	9,295	22	–	63	9,358	–	15	9,310	–	19	9,314
p2927k	25,856	666	92	3,929	29,785	80	3,298	29,154	73	3,120	28,976

(b) comparison to one-stage approach

	best results of			one-stage approach		
	two-stage approach			20s timeout		no timeout
circuit	threads	aborted	total (s)	aborted	time (s)	time (s)
p77k	4	–	5,548	–	5,454	–
p286k	2	1	3,347	1	3,456	3,497
p295k	1	1	1,107	1	1,159	1,228
p330k	4	16	2,867	32	3,208	23,475
p388k	4	2	840	2	830	1,263
p469k	4	3	20,081	120	13,139	30,815
p1522k	2	–	9,310	–	9,324	–
p2927k	4	73	28,976	87	33,758	50,812

approach, the run-time and number of aborts of the two-stage method are both considerably lower in many cases (e.g. p330k and p2927k). In the case of p469k, the two-stage approach needs 20,081 seconds as opposed to the one-stage method that needs only 13,139 seconds, but the two-stage approach leaves only 3 faults unclassified. In contrast, the one-stage method without timeout needs a total of 30,815 seconds to classify all faults. Hence, the two-stage approach offers the best compromise between number of aborts and run-time.

The two-stage test generation was also applied to CMS@ faults for gate-exhaustive testing. Hard-to-detect faults were identified for circuits p330k, p469k and p2927k by running Tiguan in single-threaded mode with a timeout of one second per fault.

Table 17
Two-stage thread-parallel CMS@-based SAT-ATPG for gate-exhaustive testing

	p330k (1st st: 25,601s, 203 abr)			p469k (1st st: 59,493s, 5,480 abr)			p2927k (1st st: 340,603s, 1,621 abr)		
	2nd stage			2nd stage			2nd stage		
threads	abr	time (s)	total (s)	abr	time (s)	total (s)	abr	time (s)	total (s)
1	70	2,087	27,688	421	23,919	83,412	284	11,666	352,269
2	38	1,550	27,151	147	10,713	70,206	260	10,212	350,815
4	36	1,246	26,847	18	5,614	65,107	258	9,651	350,254
6	38	1,196	26,797	7	6,610	66,103	246	9,272	349,875
8	30	1,055	26,656	3	6,527	66,020	246	9,533	350,136
12	28	948	26,549	1	7,119	66,612	247	10,590	351,449
16	27	898	26,499	0	8,566	68,059	232	10,133	350,736

The number of faults targeted in the first stage was of 1,166,046 for p330k (about 12% of them undetectable), 312,784 for p469k (23% of them undetectable), and 7,048,378 for p2927k (about 11% of them undetectable). For p330k, the run-time for the first stage was of 25,601 seconds, and 203 faults were aborted. For p469k, all but 5,480 faults were classified in 59,493 seconds, and for p2927k, the run-time of the first stage was of 340,603 seconds, with 1,621 aborted faults. The second stage was executed by Tiguan in multi-threaded mode using different numbers of threads between 1 and 16. All other NXP circuits are not considered, as they did not produce a sufficient number of aborts in the first stage.

Table 17 reports the number of aborts left after the second stage (columns *abr*), the run-time in seconds of the second stage (columns *time*), and the accumulated run-time in seconds of both stages (columns *total*). While using more cores always increases the ATPG quality in terms of the amount of classified faults, the run-time does not become considerably better for more than 4–6 computation threads in the case of p330k, and in the case of the other two circuit, the total run-time even starts increasing again.

In order to evaluate the two-stage approach, the experiment was repeated using multiple threads and a timeout of 20 seconds (one-stage approach with parallelism). This resulted in a total run-time of 75,072 seconds for p330k when using 16 threads, while the two-stage approach achieves the best run-time of 26,499 seconds when run with 16 threads. For p469k, 4 threads yielded the best run-time of 75,157 seconds as opposed to 65,107 seconds in two-stage mode (also 4 threads). The fact that the one-stage approach with parallelism needs more time than the two-stage approach may seem counter-intuitive at first, because the one-stage approach uses more resources. However, this is due to the thread-management overhead incurred in the one-stage approach which is unnecessary for easy-to-solve ATPG instances. This confirms the superiority of the two-stage approach.

5.2 SAT-ATPG WITH INCREMENTAL SAT SOLVING

This section introduces the fault clustering technique, a different acceleration technique for SAT-ATPG that makes use of incremental SAT solving provided by the SAT solver ANTOM. After a short review of the differences between ANTOM and MiraXT, the technique is introduced and evaluated by application to stuck-at faults and to CMS@ faults for gate-exhaustive testing.

5.2.1 THE SAT SOLVING BACK-END ANTOM

ANTOM is a new SAT solver whose development at the Chair of Computer Architecture in Freiburg was started in 2010 with the intention of implementing a SAT solver that supports incremental SAT solving as well as SAT solving with qualitative preferences. Moreover, ANTOM incorporates new SAT solving techniques that have emerged recently. ANTOM has already been successfully deployed into several test and verification tools [117, 209, 210, 207], also in the form of a #SAT solver [82, 83], and as the foundation of a QBF solver [212, 213]. Here, only some of the differences between ANTOM and MiraXT are explained. More detailed information on ANTOM can be found in [217].

In contrast to MiraXT, ANTOM supports incremental SAT solving (see Section 3.3.3) and it also accepts *assumptions*, i.e. an initial partial assignment of Boolean variables that is preserved by the solving procedure. These two aspects were used to implement the fault clustering technique that is the subject of this section. In addition, ANTOM has the ability to solve SAT problems with qualitative preferences (see Section 3.3.4), although the efficiency of preference processing is currently limited, given that the tool is still in active development. Nonetheless, this feature was successfully used to extend TIGUAN in order to support complex ATPG problems with optimisation objectives. This topic will be discussed in detail in Chapter 7.

In contrast to MiraXT, ANTOM also supports the abortion of SAT solving based on a backtracking limit, i.e. the search can be aborted after a given number of conflicts. This can be implemented more efficiently than the abortion based on timeouts, because abortion based on timeouts requires constant polling of the elapsed time.

In addition, abortion based on a backtracking limit guarantees that the results are deterministic with regard to which faults are aborted. Finally, ANTOM was also tuned for better performance on ATPG instances, but instead of the empirical approach used for MiraXT, ANTOM's internal control variables were adjusted using a dedicated parameter tuning tool [127, 3].

Aside from several minor techniques to increase the efficiency of BCP and conflict analysis, there are two major differences between ANTOM and MiraXT regarding the implementation of multi-threading and the way in which the problem is distributed among different threads, and regarding the decision strategy. An example technique employed to speed up BCP consists in storing two-literal clauses in an own database, such that these clauses can be processed first when BCP is performed, which also results in reduced memory consumption. There are also further differences. For example, while MiraXT implements only static restarts [157], ANTOM also implements a procedure introduced by the SAT solver Glucose [19] which performs restarts only when necessary. However, these implementation details are less relevant from the point of view of SAT-based ATPG, as SAT formulae derived from ATPG problems can be solved in times that are clearly below the average of the random SAT instances for which the SAT solvers were originally tuned.

In ANTOM, each computation thread manages an own clause database, which has the disadvantage of increased memory use and slightly slower communication between threads in comparison to the architecture used by MiraXT. However, ANTOM's architecture speeds up BCP, since pointers to a shared clause database do not need to be resolved any more, and since the necessity of locks is diminished. These advantages are particularly observed when the tool is started in single-thread mode.

Instead of search-space splitting, ANTOM employs multi-threading to implement the *algorithm portfolio* principle [153]. According to this principle, every thread solves the complete SAT instance, but each thread employs different algorithms or different parameter settings. The motivation for this model is that modern multi-purpose SAT solvers are employed to solve SAT instances with very different levels of hardness; hence, it is usually difficult to find one single set of parameter settings that suits all types of SAT instances. Moreover, as opposed to MiraXT, this enables ANTOM to produce deterministic results also in multi-threading mode, which makes the development and debug of applications easier. However, ANTOM's multi-threading capability is not used by TIGUAN, as its implementation had not been yet finished at the time when the work on fault clustering was being done.

The second major difference between ANTOM and MiraXT consists in the use of an improved decision strategy that eliminates the need of decay operations. Instead of periodically halving the variable counters, the counter of each variable that occurs in a new clause is increased by a factor that grows over time[25]. This strategy has the same effect as the periodic counter halving — it gives newer clauses preference over older clauses. Also, the occurrences of each variable are recorded in one single counter disregarding the polarity of each occurrence. Then, when making a decision, the chosen variable is assigned the last value that this variable has been previously assigned, instead of assigning a value depending on the difference between affirmative and negative occurrences[26]. Altogether, the combination of these improvements results in a decision strategy that causes less run-time overhead than the classic VSIDS strategy implemented in MiraXT.

5.2.2 Fault clustering

Given a combinational circuit C and a list of faults $f_1, \ldots, f_N$, traditional SAT-based ATPG (without fault dropping) generates N SAT formulae $\varphi_1, \ldots, \varphi_N$, one per fault, and solves each of them separately. Various approaches to reduce the run-time of SAT-based ATPG try to enhance the run-time of both SAT formulation and SAT solving, but in general they stick to the principle of generating and solving one formula per fault. In [244, 243], for instance, the circuit is partitioned into fan-out-free regions (FFR, see Section 2.2.2), and each FFR is represented using a BDD. Then, the SAT formula that describes the structure of the circuit is derived from the BDDs, which removes some information redundancy and allows to reuse already converted sub-formulae. However, this approach is only applicable to FFRs with a limited amount of gates due to the memory requirements of BDDs. A better approach that achieves a considerable reduction in SAT solving time is presented in [70]. Here, a SAT-solver-external clause database is used to cache original and learnt clauses corresponding to the fault-free circuit. However, this approach still has to either generate a new SAT formula or to modify an existing SAT formula for each new processed fault.

Motivated by the empirical observation that the time required for SAT generation is not only non-negligible, but also often higher than the average SAT solving time (see Section 4.6), the proposed fault clustering approach attempts to speed up ATPG by substantially reducing the number of required SAT formulation runs and completely eliminating the need to modify existing SAT formulae. This is done at the expense of

[25] This technique was introduced by MiniSat [80, 1].

[26] This technique was introduced by RSAT [178]. The intuition behind this type of variable assignment is that the SAT solver can enter parts of the search tree that have been explored before. Hence, the solver will benefit from previously learnt conflicts.

larger and slightly more complex SAT instances. However, incremental SAT solving is used in order to prevent excessive growth of the average SAT solving time.

In SAT-ATPG with fault clustering, the input fault list is divided into sub-sets of faults called *fault clusters*. Let one such cluster be composed of faults $f_1, \ldots, f_n$. Instead of generating *n single-fault SAT formulae* $\varphi_1, \ldots, \varphi_n$, fault-clustering-based ATPG generates only one *combined SAT formula* $\varphi_{1:n}$ that represents a circuit into which the faults $f_1, \ldots, f_n$ can be injected depending on the values of *control variables* $X_1, \ldots, X_n$. $\varphi_{1:n}$ is constructed such that, for all $i = 1, \ldots, n$, the single-fault SAT formula φ_i is semantically equivalent to

$$\varphi_{1:n} \wedge X_i \wedge \bigwedge_{j \neq i} \neg X_j.$$

The combined SAT formula $\varphi_{1:n}$ needs to be generated and passed to the SAT solver only once. Hence, not only the time needed to traverse the circuit in order to generate clauses is saved, but also the time that the SAT solver requires to insert new clauses into its internal database and to set up its internal data structures.

The combined SAT formula $\varphi_{1:n}$ is then solved once for each fault in the cluster, however using different assumptions each time. In order to solve the combined SAT formula $\varphi_{1:n}$ such as to find a test for f_i, the set of assumptions

$$X_i = 1,\ X_1 = \cdots = X_{i-1} = X_{i+1} = \cdots = X_n = 0$$

is passed to the SAT solver, meaning that only f_i is injected into the circuit. The implications that result from the combined SAT formula and from these assumptions are equivalent to φ_i by construction of $\varphi_{1:n}$. Hence, solving the combined SAT formula given these assumptions renders a valid test pattern for f_i if one exists, or it proves the undetectability of f_i if the assumptions lead to a conflict.

Between two calls of the SAT solver's solving routine using two different sets of assumptions, the SAT solver is not reset. This means that all global implications and conflict clauses learnt by the SAT solver during previous solving runs are used to speed up the solving of the combined SAT formula under the current set of assumptions. Therefore, clustering results in a speed gain even as the combined SAT formula $\varphi_{1:n}$ is larger and on average harder than each single-fault formula $\varphi_1, \ldots, \varphi_n$.

The principle for the construction of $\varphi_{1:n}$ is as follows. Let the fault cluster to be encoded be composed of n CMS@ faults $f_1, \ldots, f_n$, and let $X_1, \ldots, X_n$ be the chosen control variables, where each X_i being set to logic 1 is equivalent to the injection of f_i into the circuit, for $i = 1, \ldots, n$. Let l be a line in the circuit, and let $K^l_{i,1}, \ldots, K^l_{i,m_i}$

be the clauses in the single-fault formula φ_i that describe l's behaviour in the circuit C^{f_i}, i.e. the circuit that is affected by fault f_i. The behaviour of the line is either that of a normal line, or that of one of f_i's victims, or that of one of f_i's aggressors. Then, for each $k = 1, \ldots, m_i$, the clause $\{\neg X_i\} \cup K^l_{i,k}$ is added to the combined SAT formula $\varphi_{1:n}$. If fault f_i is injected into the circuit by setting X_i to 1, $\neg X_i$ evaluates to 0 and $K^l_{i,k}$ remains in the combined SAT formula after the propagation of the constraints introduced by the assumption $X_i = 1$. In contrast, if fault f_i is removed from the circuit by setting X_i to 0, that assignment immediately satisfies $\{\neg X_i\} \cup K^l_{i,k}$ and the SAT solver's solving routine removes this clause from the SAT formula. Hence, line l's behaviour expressed by $K^l_{i,k}$, which is valid only if f_i is present, is subsequently ignored by the SAT solver. This step has to be repeated for each $i = 1, \ldots, n$ and for each line l. From the point of view of the implementation, this principle can bear some redundancy. Hence, in order to eliminate redundant sub-formulae in the combined SAT formula, behaviour that is the same in different circuit instances is modelled only as many times as necessary.

This technique was implemented in a way that would preserve the underlying structure of the tool such as to benefit from optimisation techniques previously introduced into Tiguan. Figure 21 illustrates an example in which two CMS@ faults $f_1 : v$ s-a-1 and $f_2 :$ if $[a = 1]$ w s-a-0 form a fault cluster. As in the SAT formulation without clustering[27], the first step consists in colouring the circuit, which is done according to the same criteria as without fault clustering, but with the difference that the output cones of all victims and the input cones of all aggressors that belong to any fault in the cluster need to be taken into consideration. Hence, the coloured circuit area and thus the resulting SAT formula will be larger than in the case that single-fault SAT formulae are constructed. For this reason, it is important to consider which faults can be grouped into clusters such that the size of the combined SAT formula is not significantly larger than the size of the various single-fault SAT formulae. Since SAT's worst-case complexity is exponential in the number of variables, clustering faults whose influence regions have only a small area in common can result in a combined SAT formula that is too large and hence too hard to solve. Different strategies for the grouping of faults into clusters are evaluated later.

In order to allow Tiguan to assign Boolean variables to circuit lines and to construct clauses in the same efficient fashion as the algorithm without fault clustering would do, the internal data structure that is used to represent the circuit was extended such that each node can store information regarding which colour it would have in each

[27] The SAT formulation without clustering was introduced in detail in Section 4.4.

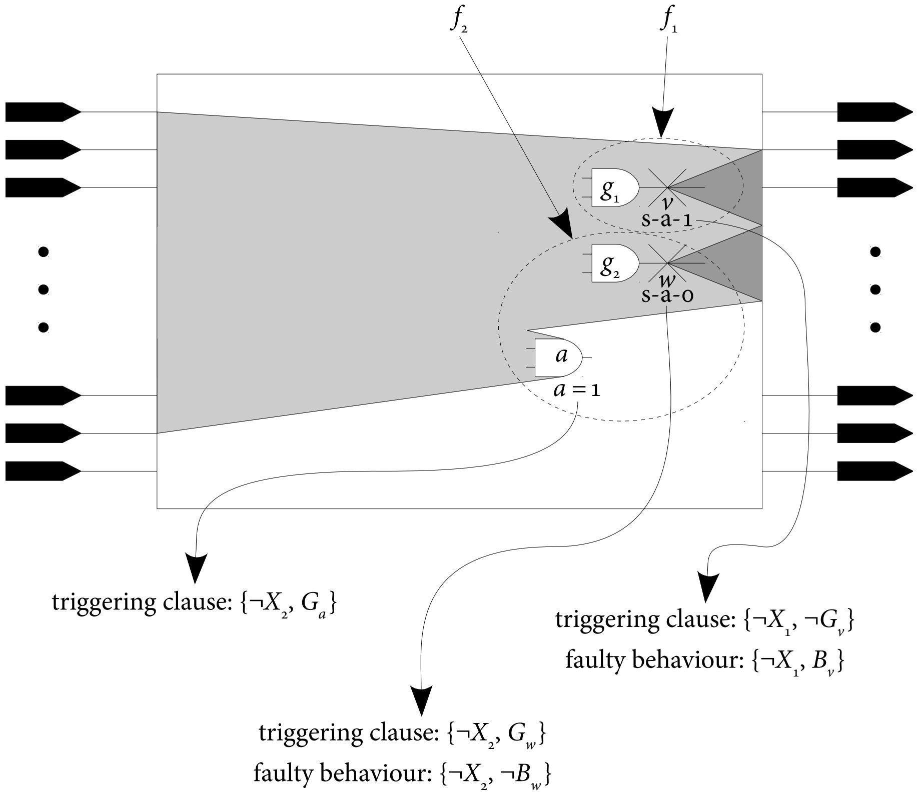

f_2
f_1
g_1
v
s-a-1
g_2
w
s-a-0
a
$a = 1$
triggering clause: $\{\neg X_2, G_a\}$
triggering clause: $\{\neg X_1, \neg G_v\}$
faulty behaviour: $\{\neg X_1, B_v\}$
triggering clause: $\{\neg X_2, G_w\}$
faulty behaviour: $\{\neg X_2, \neg B_w\}$

Figure 21. Generation of the combined SAT instance for a fault cluster

circuit instance, and whether it would be a victim or an aggressor with respect to which fault. According to these data, the correct type of clauses can be constructed for each node, and the combined SAT formula is constructed on the fly without the need to first construct the single-fault SAT formulae. In the example from Figure 21, fault f_1 is a CMS@ fault with only one s-a-1 victim v. Hence, this fault can only be excited if line v is set to 0 in the fault-free case, which results in the triggering clause[28] $\{\neg G_v\}$. But this clause shall be valid only when solving the combined SAT formula for fault f_1, i.e. when the SAT solver is given the assumption $X_1 = 1$. Hence, instead of $\{\neg G_v\}$, the clause $\{\neg X_1, \neg G_v\}$ is inserted into the combined SAT formula. Analogously, line v's faulty behaviour under the presence of f_1 is represented by the clause $\{\neg X_1, B_v\}$, which has an effect on the SAT solving only when X_1 is set to 1. In the same way, all clauses that represent the behaviour of lines in v's output cone

[28] Recall that G-variables model the line's logic value in the fault-free circuit, while B-variables model the line's logic value in the fault-affected circuit (see Section 4.4).

under the presence of f_1 have to be given the additional control literal $\neg X_1$, while the clauses that represent those lines' behaviour in the fault-free case are the same as in the single-fault SAT formula.

The second fault in this example, f_2, is a CMS@ fault with one s-a-0 victim w that is activated only if the aggressor line a is set to 1. Similarly to the f_1-case, the clauses that excite the victim w and that represent its faulty-case behaviour need to be extended by the control literal $\neg X_2$. Also, the fault activation condition $a = 1$ shall be enforced only when the control variable X_2 is set to 1. Thus, the clause $\{\neg X_2, G_a\}$ is inserted into the combined SAT formula.

Finally, though not shown in Figure 21 for better legibility, all clauses that relate D-variables[29] to B-variables that are bound to the control variables X_1 or X_2, need to inherit the respective control literals. This also applies to the D-variables assigned to coloured primary outputs; these D-variables are used to generate the triggering clause that demands that at least one primary output must show a fault effect.

An important aspect of the fault clustering technique and of the proposed principle for the construction of $\varphi_{1:n}$ is that the application of the technique to clusters of size 1 poses no overhead with respect to the original TIGUAN algorithm introduced in Section 4.4. Due to the chosen implementation, the SAT formulation procedure needs to insert the only control literal $\neg X_1$ into few selected clauses, for which no additional traversing of the circuit is needed, as the necessity of control literals at each node is determined on the fly by the colouring procedure. Also, the formula $\varphi_{1:1}$ together with the assumption $X_1 = 1$ is not harder to solve than φ_1 because the assignment $X_1 = 1$ is handled by the SAT solver as a non-backtrackable first-level decision, which causes no search overhead.

5.2.3 Experimental evaluation

In order to evaluate the quality of the fault clustering technique, test patterns for single-stuck-at faults and for CMS@ faults for gate-exhaustive testing were computed for ISCAS'85 benchmark circuits and for the combinational cores of ITC'99 and NXP circuits. All measurements were performed on a 2.3 GHz AMD Opteron 64-bit computer with 64 GB RAM. In all experiments, an unlimited time budget was assigned to every processed fault; hence, no faults were aborted and the total run-times are also influenced by the time needed for very hard instances.

[29] Recall that D-variables model whether a gate's output displays a fault effect (see Section 4.4).

Table 18

SAT-ATPG with fault clustering for stuck-at faults — without fault dropping

			SAT formulation		SAT solving		
circuit	faults	clustering	runs	time (s)	runs	time (s)	total time (s)
c5315	5,350	1	5,350	0.0015	5,350	0.0002	9
		10	535	0.0056	5,350	0.0004	6
		20	268	0.0112	5,350	0.0008	8
		50	107	0.0379	5,350	0.0039	25
		ffr	929	0.0042	5,350	0.0002	5
c6288	7,744	1	7,744	0.0089	7,744	0.0025	88
		10	775	0.0189	7,744	0.0045	50
		20	388	0.0343	7,744	0.0067	66
		50	155	0.0922	7,744	0.0140	124
		ffr	1,488	0.0151	7,744	0.0039	54
c7552	7,550	1	7,550	0.0041	7,550	0.0004	34
		10	755	0.0095	7,550	0.0008	14
		20	378	0.0221	7,550	0.0021	25
		50	151	0.0631	7,550	0.0080	71
		ffr	1,408	0.0065	7,550	0.0004	13
b17c	68,207	1	68,207	0.0320	68,207	0.0094	3,487
		10	6,821	0.0713	68,207	0.0120	2,235
		20	3,411	0.1056	68,207	0.0175	2,434
		50	1,365	0.2757	68,207	0.0482	4,711
		ffr	9,155	0.0595	68,207	0.0063	1,826
b18c	206,812	1	206,812	0.0803	206,812	0.0129	25,912
		10	20,682	0.1459	206,812	0.0306	19,524
		20	10,341	0.2080	206,812	0.0442	21,674
		50	4,137	0.4475	206,812	0.0864	30,285
		ffr	28,395	0.1389	206,812	0.0158	16,352
b20c	35,731	1	35,731	0.0210	35,731	0.0037	972
		10	3,574	0.0608	35,731	0.0127	816
		20	1,787	0.1120	35,731	0.0227	1,163
		50	715	0.2165	35,731	0.0438	1,833
		ffr	5,090	0.0526	35,731	0.0049	600
b21c	36,058	1	36,058	0.0217	36,058	0.0042	1,023
		10	3,606	0.0593	36,058	0.0122	787
		20	1,803	0.1026	36,058	0.0224	1,137
		50	722	0.2554	36,058	0.0519	2,207
		ffr	5,187	0.0501	36,058	0.0048	567
b22c	51,341	1	51,341	0.0231	51,341	0.0037	1,552
		10	5,135	0.0731	51,341	0.0146	1,493
		20	2,568	0.1148	51,341	0.0247	1,901
		50	1,027	0.2918	51,341	0.0637	3,942
		ffr	7,397	0.0555	51,341	0.0042	932

In a first experiment, ATPG without fault clustering and ATPG with fault clustering in different configurations was applied to each considered circuit. The results are listed in Table 18, which includes only the larger ISCAS'85 and ITC'99 circuits. For an accurate evaluation of the real impact of reducing the number of SAT formulation runs on the total run-time, no fault simulation was performed. For this reason, industrial circuits were not considered in this experiment.

The column labelled *faults* lists the number of processed faults (all stuck-at faults after the removal of locally equivalent faults). Column *clustering* indicates what type of fault clustering applies to the corresponding line. A number n means that faults were grouped in clusters of fixed size n. In this case, the input fault list was sorted according to the topological order of the victim lines, with faults affecting the primary outputs of the circuit at the beginning of the list and faults affecting the primary inputs at the end of the list (*RTOP sorting*, see Section 6.3). Then, the sequence of faults was partitioned into sequences of n faults to form the clusters, i.e. faults $f_1, \ldots, f_n$ constituted the first cluster, $f_{n+1}, \ldots, f_{2n}$ constituted the second cluster, and so on. Note that *clustering* = 1 stands for clusters of size 1, i.e. for traditional SAT-ATPG without fault clustering. In contrast, *clustering* = *ffr* means that fault clusters of variable size were used, where each fault cluster contained all faults affecting the gates of a fan-out-free region. The columns labelled *runs* and *time* list the number of calls of the SAT formulation and the SAT solving procedures, and the average run-time in seconds per run, respectively. The last column of the table lists the total run-time in seconds.

When no clustering is applied (*clustering* = 1), the number of SAT formulation runs equals the number of SAT solving runs, as a SAT formula is generated and solved separately for each processed fault. The average run-times for the SAT formulation and SAT solving procedures measured in this case confirm the observation that served as motivation for the development of fault clustering — the average time needed to generate a formula for SAT-ATPG is higher than the average SAT solving time (see Table 5).

As expected, the average SAT solving time increases with growing cluster sizes. However, the amount of time saved by calling the SAT formulation routine significantly less often results in an average total-time reduction of 24.6% for clusters of size 10. However, the reduction decreases for clusters of size 20, and clusters of size 50 even increase the total run-time since the SAT solving time grows substantially for that cluster size.

However, not only the cluster size influences the quality of fault clustering. The approach that groups all faults of an FFR into one cluster achieves the best results, even though clusters with up to 200 faults were observed for the circuits in Table 18,

Table 19
SAT-ATPG with fault clustering for stuck-at faults — with 64-bit fault dropping

			SAT formulation		SAT solving		total	
circuit	faults	clustering	runs	time (s)	runs	time (s)	time (s)	red (%)
p35k	67,733	none	11,297	0.1903	11,297	0.1126	3,483	
		ffr-based	1,340	0.1874	10,254	0.0904	1,207	65.3
p77k	120,348	none	12,282	0.1381	12,282	0.4153	7,214	
		ffr-based	3,273	0.1834	12,488	0.2883	4,527	37.2
p81k	204,174	none	23,105	0.0688	23,105	0.0024	1,746	
		ffr-based	5,667	0.0917	20,593	0.0011	647	62.9
p89k	150,538	none	11,445	0.0587	11,445	0.0024	734	
		ffr-based	2,620	0.0959	13,518	0.0033	344	53.1
p141k	282,428	none	11,139	0.2859	11,139	0.1042	4,397	
		ffr-based	3,843	0.2705	10,471	0.0535	1,643	62.6
p267k	366,871	none	12,482	0.1792	12,482	0.0059	2,396	
		ffr-based	2,079	0.2747	12,855	0.0143	852	64.4
p269k	369,055	none	12,467	0.1804	12,467	0.0058	2,405	
		ffr-based	2,082	0.2821	13,002	0.0140	861	64.2
p295k	472,022	none	26,888	0.1707	26,888	0.0075	4,994	
		ffr-based	4,314	0.3258	24,031	0.0215	2,297	54.0
p330k	540,758	none	26,616	0.2458	26,616	0.3140	15,109	
		ffr-based	4,297	0.3826	26,462	0.4457	13,843	8.4
p469k	142,751	none	2,585	0.3660	2,585	17.2661	48,055	
		ffr-based	1,152	0.3937	2,483	7.9524	21,155	56.0

and with up to 2,500 faults for the circuits in Table 19. Since all faults in such a cluster share the same area of influence, they produce very similar single-fault SAT formulae. Hence, a large amount of conflict knowledge can be shared between different SAT solving runs of the combined SAT formula. In fact, thanks to incremental SAT solving, the average SAT solving time using FFR-based clustering is only negligibly higher than in the case without clustering, and in some cases it is even better (e.g. b17c).

In a second experiment, the fault clustering technique was applied to harder NXP circuits. The three largest NXP circuits, p951k, p1522k and p2927k, were excluded due to memory limitations. Given the size of NXP circuits and the very large amount of targeted faults which make ATPG without fault simulation infeasible, fault clustering was combined with fault dropping with a simulation width of 64, i.e. actual fault simulation was performed in PPSFP (parallel-pattern single-fault propagation, see Section 2.6) fashion whenever 64 new unique test patterns had been collected. If a simulation run had to be performed while a cluster was being processed, faults in the current cluster that were detected by simulation were dropped in order to avoid unnecessary SAT solving runs, but the SAT formulation for the current cluster was not repeated.

Based on the observations of the first experiment, according to which FFR-based fault clustering is the most effective strategy for fault grouping, only the FFR-based clustering technique was applied and compared to SAT-ATPG without clustering. The results are summarised in Table 19. The columns in this table have the same meaning as in Table 18, but there is an additional column labelled *red* which lists the reduction of total run-time with respect to SAT-ATPG without clustering. The reduction is defined by

$$\text{reduction} = \frac{\text{time without clustering} - \text{time with clustering}}{\text{time without clustering}} \cdot 100\%.$$

In contrast to ISCAS'85 and ITC'99 circuits, the average SAT solving times per run with fault clustering are better than without clustering in most cases, and especially large differences can be observed for circuits p77k and p469k which are both known as particularly hard benchmarks. These data show that the solving of large and hard SAT instances benefits greatly from the application of incremental SAT solving. As in the case of ISCAS'85 and ITC'99 circuits, the total run-time also benefits from the reduced number of SAT formulation runs, which in this case is even lower due to the use of fault simulation. Since larger FFRs are processed first, many clusters corresponding to smaller FFRs are completely detected by simulation, which means that the combined SAT formula for those FFRs does not need to be generated any more. The combination of both a reduced number of SAT formulation runs and faster SAT solving due to incremental learning results in an average reduction of total run-time of 47.7%, and a reduction of more than 60% for half the circuits.

The same experiment was also applied to CMS@ faults for gate-exhaustive testing. Such faults result in SAT instances that are harder than SAT instances derived from stuck-at faults, because gate-exhaustive testing imposes very specific conditions on the gate that drives the fault site. Table 20 shows the results of this experiment. Here, the average SAT solving time per run is higher when fault clustering is applied, which can in part be attributed to the increased size of the FFR-based clusters in comparison to FFR-based clusters of stuck-at faults (gate-exhaustive testing targets more faults per gate), but the difference is only minimal. Combined with the high

Table 20

CMS@-based SAT-ATPG with fault clustering for gate-exhaustive testing — with 64-bit fault dropping

			SAT formulation		SAT solving		total	
circuit	faults	clustering	runs	time (s)	runs	time (s)	time (s)	red (%)
p35k	150,898	none	32,299	0.2259	32,299	0.0614	9,462	
		ffr-based	4,347	0.1577	28,599	0.1700	5,746	39.3
p77k	254,176	none	50,454	0.1184	50,454	0.1141	13,531	
		ffr-based	8,901	0.1783	51,097	0.0994	8,314	38.6
p81k	404,348	none	99,984	0.0577	99,984	0.0012	6,387	
		ffr-based	16,314	0.1063	94,526	0.0008	3,357	47.4
p89k	306,680	none	46,974	0.0427	46,974	0.0009	2,117	
		ffr-based	6,741	0.1144	47,744	0.0031	1,177	44.4
p141k	577,922	none	66,080	0.1777	66,080	0.0241	13,417	
		ffr-based	21,438	0.2041	65,166	0.0151	5,598	58.3
p267k	850,024	none	86,690	0.1373	86,690	0.0013	12,177	
		ffr-based	22,819	0.2382	85,680	0.0203	7,797	36.0
p269k	854,364	none	86,843	0.1093	86,843	0.0010	9,717	
		ffr-based	22,807	0.2591	85,844	0.0184	8,039	17.3
p295k	986,740	none	136,946	0.1230	136,946	0.0018	17,367	
		ffr-based	22,048	0.3210	135,318	0.0388	13,732	20.9
p330k	1,166,046	none	165,529	0.2187	165,529	0.1091	54,828	
		ffr-based	35,806	0.2644	159,423	0.1291	32,407	40.9
p469k	312,784	none	71,997	0.4954	71,997	1.0285	113,155	
		ffr-based	7,014	0.8865	71,872	0.9144	73,124	35.4

volume of saved SAT formulation time, the average reduction of total time amounts to 36.8% and is as high as 58.3% for some circuits (p141k).

5.3 Conclusions

The study presented in the first part of this chapter evaluated the scalability of state-of-the-art SAT-based ATPG software to multi-core systems. The data showed that the algorithm performs well on mid-size systems with around 4–6 available cores. Providing more resources slightly increases the quality in terms of the amount of classified faults, but no further speed-up can be registered, and in some cases run-times even increase. The primary reason for this is memory congestion that occurs due to the large amount of clauses simultaneously learnt by individual computation threads, and due to the shared-memory architecture of the SAT solver. Future research maintaining this SAT solving architecture should focus on adaptive strategies to select which learnt clauses are written back into the shared memory such that memory congestion is avoided. Another option consists in reconsidering earlier multi-threading approaches that employed fault parallelism, i.e. different parts of the fault list were assigned to different computation nodes [88], but taking into account that the technology has changed.

The study also evaluated the different impact of thread-parallel SAT-ATPG on easy and hard-to-detect faults. It determined that thread parallelism is mostly ineffective for the processing of easy-to-solve ATPG instances, while very hard instances benefit strongly from increased resources. In consequence, a two-stage technique was developed, where all faults are targeted without thread parallelism in the first stage. In the second stage, only faults aborted in the first stage are targeted using thread parallelism. The approach offers the best compromise between test quality and run-time in comparison to one-staged approaches with and without thread parallelism.

The second part of this chapter presented a fault clustering technique that makes use of incremental SAT solving. The principle of this technique consists in generating one single SAT formula for a set of faults (a cluster) and solving the same formula under specific assumptions once for each fault. It was determined that the strategy by which faults are grouped into clusters has a major influence on the speed gain. A strategy based on fan-out-free regions allowed TIGUAN to classify all stuck-at faults in large industrial circuits in 47.7% less run-time.

One possible direction for future research consists in combining fault clustering with thread parallelism. The SAT solver ANTOM, which is used to implement fault clustering in TIGUAN, employs multi-threading to implement the algorithm portfolio principle, i.e. each computation thread solves the complete SAT instance independently using different algorithms and different SAT solving parameters. A modification of ANTOM that would allow to solve the same instance in parallel using different sets of assumptions would enable the parallel generation of several test patterns corresponding to one single fault cluster.

Even as the FFR-based clustering method achieved very good results, the best way of clustering faults is in general not known in advance. Hence, an alternative approach to combine clustering and thread-parallel processing would consist in creating several clusters and processing them in parallel, and then choosing the cluster whose processing is finished first.

6

SAT-based ATPG with dynamic compaction

After the discussion of techniques to enhance the run-time efficiency of TIGUAN, also the issue of test set compactness is considered, given that the pattern count of SAT-based ATPG is generally larger than that of structural ATPG. This chapter introduces a dynamic compaction method[30] that was developed for dedicated incorporation into TIGUAN. The method overcomes the over-specification of SAT-based-generated test patterns and uses the specific interfaces offered by the CMS@-based framework in order to achieve maximum efficiency. After an initial evaluation of the method's performance on large industrial circuits, the last part of the chapter presents an enhancement of the basic technique and analyses the impact of fault list pre-sorting on its performance.

AUTHOR'S CONTRIBUTION — The author's contribution consisted in the evaluation of TIGUAN's initial performance regarding the amount of generated patterns, and the development, evaluation and enhancement of a dynamic compaction technique that exploits the possibilities offered by the existing CMS@-based framework.

Parts of the work covered in this chapter have been published in [C13] (see author's publications on pages 223–226).

[30] See Section 2.7.2 for an introduction to compaction.

6.1 Introduction

While the run-time efficiency of SAT-based ATPG makes it an attractive alternative to structural approaches, in particular for the processing of hard-to-detect and undetectable faults, one weakness of early SAT-based ATPG methods is their relatively high pattern count, which results largely from the over-specification of the generated patterns.

Structural ATPG generally searches for a test pattern starting at the fault site and moving towards the primary inputs. Hence, only inputs necessary for fault detection are assigned 0 or 1-values, while all other inputs are left unspecified. Also, since structural algorithms perform the search based directly on the net list, the implementation of heuristic techniques for more compact test sets is often easier. For example, the tool COMPACTEST [185] modifies the objectives of line justification dynamically in order to allow new patterns to detect more faults not previously detected. Another good example is the subscripted D-Algorithm [170] which attempts to sensitise multiple paths simultaneously such as to generate single patterns that detect more faults. Such approaches, however, are not directly applicable to SAT-ATPG, as the good performance of SAT-ATPG on undetectable and hard-to-detect instances stems from the liberty the SAT solver has to apply SAT-specific pruning techniques that are not aware of the structure of the original problem. Also, contrary to the DPLL-Algorithm, many modern SAT solvers specify all Boolean variables, because they assign a fixed or random value to all Boolean variables at the beginning, and then modify the values of single variables until resolving all conflicts, which results in equally valid solutions of the SAT problem but is usually more efficient (see Section 5.1.1).

For this reason, early SAT-ATPG approaches [68] rely on test relaxation after the test pattern has been generated. While all primary inputs outside of the fault site's influence region are trivially left unspecified, also many primary inputs within the influence region are relaxable since the justification of controlling values on a gate's output (non-controlling value if the gate is inverting) requires the justification of the gate's controlling value on only one of its inputs, while all other inputs can be left unassigned as long as they are not necessary to satisfy other justification tasks. However, relaxable inputs within the influence region of the fault site cannot be directly identified using SAT-based ATPG. In Tiguan, an input-output-cone analysis is performed immediately after the SAT solver computes a model of the SAT formula. This analysis determines which of the primary outputs towards which the generated test pattern propagates a fault effect requires the smallest number of primary inputs to be assigned a specified value, and relaxes all primary inputs that do not belong to the input cone of the identified optimal primary output (see Section 4.5). In [68], a technique that consists in analysing local justification conditions is used to identify further primary inputs that can be relaxed. However, this technique, which is already widely used in structural ATPG, can be very time-consuming when combined with SAT-ATPG, as SAT-ATPG does not have additional information to aid such an analysis, whereas structural ATPG algorithms collect that kind of information without negative effect on their performance. Although high percentages of unspecified bits are reported for industrial circuits in [68], the efficiency of this technique may not be high, as faults that belong to the same cone are still likely to result in test patterns with conflicting assignments to the circuit inputs. Even though these inputs can be relaxed without sacrificing detectability, the method from [68] does not handle such situations. Furthermore, [68] quotes pattern counts only after the execution of static compaction, so that it is impossible to evaluate the actual impact of their relaxation technique on the compaction quality.

One attempted approach to utilise SAT-specific techniques for test relaxation consisted in modifying Tiguan's SAT solving engine MiraXT, such that the relaxation of the SAT solutions would be carried out by the SAT solver instead of externally. However, the percentage of variables that could be relaxed without compromising MiraXT's run-time was very low and, in all observed cases, none of the relaxed Boolean variables corresponded to primary inputs, but to internal lines of the circuit (see Section 4.5).

Given the difficulty of modifying SAT-ATPG in a way that would render more relaxed test patterns without significantly compromising the run-time efficiency of the test generation progress, the author of this thesis developed a dynamic compaction technique dedicated to the application with SAT-ATPG, a technique that does not rely on the amount of unspecified bits to achieve good results.

The generic dynamic compaction approach [99] consists in generating a test pattern p_1 for a *primary target fault* f_1. Then, a *secondary target fault* f_2 is chosen, and a test pattern p_2 is generated for f_2 under the condition that the circuit inputs that have been assigned to specified values by p_1 be assigned to the same values by p_2. Hence, if p_2 exists, it detects both f_1 and f_2 and p_1 can be dismissed. This process can be repeated for further secondary targets until the percentage of unspecified values in the test pattern is too low to allow the consideration of more targets.

Similarly, the technique proposed in this chapter targets *fault groups* which are enlarged consecutively. Whenever a pattern is generated for a fault f, necessary assignments to the circuit lines are extracted from Tiguan's internal data structures. Then, the procedure attempts to add another fault to the current group by generating a test for the new fault while enforcing that the assignments for detecting the other faults in the group are not violated. The approach differs from dynamic compaction for structural ATPG in that it fixes internal circuit lines instead of primary inputs, and in that it utilises the CMS@FM to efficiently handle the set of fixed logic values.

6.2 The dynamic compaction procedure

Tiguan's dynamic compaction procedure is outlined in Algorithm 5. Given a list of CMS@ faults, the procedure tries to construct *fault groups*, i.e. collections of faults for which a common test pattern exists, and to generate a pattern for each group. Since it has been reported earlier, e.g. in [185], that the ordering of the fault list has an influence on the quality of dynamic compaction, the procedure first sorts the fault list. Details on sorting are given in Sections 6.3 and 6.4.

Then, fault groups are constructed dynamically as the processing of the fault list progresses. New generated patterns are not directly inserted into the final test set P, but stored in a variable p_{group} which stands for the test pattern that detects all faults in the currently *open* fault group. With each new fault that is successfully added to the open fault group, p_{group} is replaced by a test pattern that detects all faults in the open fault group, including the new fault. p_{group} is inserted into the final test set only when the current fault group is *closed*, i.e. when the procedure decides to insert no further faults into the fault group and to open a new fault group. The value assignments that guarantee the detection of all faults in the currently open fault group are recorded in the set FGA (*fault group assignments*). These three data structures (P, p_{group} and FGA) are initialised in lines 3–5. The initial value of p_{group} is the special value *nil* which stands for an invalid or non-existent test pattern.

The main part of the ATPG procedure iterates over the fault list. In each iteration (lines 6–28), the algorithm selects a fault f from the fault list. However, the selected fault is not directly removed from the fault list because, if the algorithm determines that the fault is not compatible to the current fault group, i.e. that no pattern exists which is able to detect f in addition to the faults in the current fault group, then f needs to be processed again in order to determine whether f's incompatibility is due to contradicting detection conditions between f and the open fault group, or due to f's possible undetectability.

After f has been selected, a new fault f' is constructed (line 8) which is composed of all victims and all aggressors of f, and of additional aggressors that specify the assignment conditions currently recorded in FGA. If FGA is empty, which happens when f is the first fault of a new fault group, then f' equals f.

Algorithm 5
SAT-based ATPG with dynamic compaction

```
Inputs: CMS@ fault list $F$
Output: compact test set $P$
ATPG-WITH-DYNAMIC-COMPACTION($F$) {
    sort $F$
    $P := \varnothing$                                         ▷ initialisation
    $p_{\text{group}} := nil$
    $FGA := \varnothing$
    while $F$ is not empty do {
        $f :=$ first fault in $F$
        $f' :=$ EXTEND-FAULT-DESCRIPTION($f, FGA$)
        $p :=$ GENERATE-TEST-PATTERN($f'$)
        if $p = nil$ and $p_{\text{group}} = nil$ then {          ▷ $f$ is undetectable or aborted
            if SAT solving has been aborted then {
                classify $f$ as aborted
            } else {
                classify $f$ as undetectable
            }
            remove $f$ from $F$
        } else if $p = nil$ and $p_{\text{group}} \neq nil$ then {              ▷ $f$ is incompatible
            $q :=$ MERGE($P, p_{\text{group}}$)
            FAULT-DROPPING($q$)
            $p_{\text{group}} := nil$
            $FGA := \varnothing$
        } else {                    ▷ current fault group successfully enlarged by $f$
            $p_{\text{group}} := p$
            classify $f$ as detectable and remove it from $F$
            extract $f$'s detection conditions from the Boolean solution
            add extracted conditions to $FGA$
        }
    }
    MERGE($P, p_{\text{group}}$)
    ROFS-OR-FLROFS($P$)
    return $P$
}
```

Then, SAT-based test pattern generation is performed for f' (line 9), i.e. a SAT formula is generated and passed to the SAT solver, and a test pattern is extracted from the model computed by the SAT solver if the formula is found to be satisfiable. Note that the returned test pattern p does in general not need to be relaxed in order to facilitate compaction, because the conditions for the detection of f' that will be passed to the next processed fault are not extracted from the test pattern but from assignments to internal circuit lines. However, the basic input-output-cone analysis presented in Section 4.5 is performed by the GENERATE-TEST-PATTERN-procedure in order to facilitate the later *merging* of the group test pattern p_{group} into the final test set. If the GENERATE-TEST-PATTERN-procedure is not able to generate a test pattern, either because f' is undetectable, or because the SAT solving is aborted due to a timeout or due to a reached backtracking limit, then GENERATE-TEST-PATTERN returns the value *nil*.

After invoking the test generation procedure, the algorithm distinguishes between three cases. In the first case (lines 10–16), f is the first fault of a new fault group ($p_{\text{group}} = nil$) and the test generation procedure has not been able to generate a test pattern for f' ($p = nil$), which means that f is either undetectable or that the SAT solver has aborted due to a timeout. In this case, the fault is classified accordingly and removed from the fault list (hence excluded from further processing).

In the second case (lines 17–21), f is not the first fault of a new fault group ($p_{\text{group}} \neq nil$) but the test generation procedure has not been able to generate a test pattern for f' ($p = nil$), which means that f is either incompatible to the current fault group or that the SAT solver has aborted due to a timeout. In this case, the currently open fault group is closed, i.e. the test pattern p_{group} which detects all faults in the currently open fault group is added to the final test set, and p_{group} and FGA are set to *nil* and $\varnothing$, respectively, thus making the algorithm interpret the next processed fault as the first fault of a new fault group. Note that f is not removed from the fault list; hence, it becomes the next processed fault. Since the unsatisfiability of f' does not imply the unsatisfiability of f, f has to be reprocessed as the first fault of a new group in order to determine the reason for the incompatibility between f and the fault group that is being closed.

Adding the fault group test pattern p_{group} to the final test set (line 18) is implemented as a merging operation. That means that p_{group} is merged with a compatible test pattern p' already in P, i.e. p' is replaced by $p' \cap p_{\text{group}}$. As explained in Section 2.7.2, finding an optimal test pattern p' for intersection with p_{group} is a computationally complex problem. Moreover, in contrast to static compaction, where all test patterns are known in advance, the merging operation executed here does not know what test patterns will be generated after the current p_{group}. For this reason, no sophisticated techniques for the search for p' are employed, and p' is simply chosen as the first test pattern in P found to be compatible to p_{group}. If no compatible test pattern p' exists in P, p_{group} is added to P without merging. In the first case, the MERGE-procedure returns the pattern $q := p' \cap p_{\text{group}}$; in the second case, it returns $q := p_{\text{group}}$. Hence, in both cases, q is the newest pattern that makes part of P. Then, the FAULT-DROPPING-procedure performs fault simulation with the new pattern q. Fault dropping can only be performed with patterns that become part of the final test set. Otherwise, a misclassification of faults detected by simulation could occur. Consequently, fault dropping is performed only when a fault group is closed. Note that, although the pattern q can be modified by future merging operations, performing fault dropping with the current version of q is permissible, as those future merging operations will not modify the specified values of q, i.e. they will not reduce q's fault coverage.

In the last case (lines 22–26), where the test generation procedure has been successful, p_{group} is replaced by the new generated test pattern p. Due to the construction of f', p is guaranteed to detect not only f, but also all faults in the current fault group that are detected by the old version of p_{group}. Hence, The old value of p_{group} is discarded without inclusion into the final test set.

After classifying f as detectable and removing it from the fault list, the dynamic compaction procedure extracts a compact set of value assignments that are necessary to detect f. For CMS@ faults with exactly one victim line, these assignments are the following:

- fault excitation assignment — for the s-a-b_v victim line v, the assignment of v to the value $\neg b_v$;
- fault activation assignments — for each aggressor line a associated to a value b_a, the assignment of a to the value b_a;
- fault propagation assignments — for each gate whose output port's D-variable has been set to 1 by the computed Boolean solution (i.e. a fault effect has been propagated through that gate), the assignment of all off-path inputs of the gate to the gate's non-controlling value.

An example is shown in Figure 22. Assume that the currently processed fault is fault $f_1 : a$ s-a-0, and that the generated test pattern p propagates the fault effect through the NOR gate g with inputs a and b, and through the AND gate k with inputs g, c and d. In this case, the D-variables assigned to the output ports of g and k (D_g and D_k, respectively) must have the value 1, by construction of the SAT formula; also, D_h, the D-variable assigned to gate h has the value 0, since no fault effect is propagated through h. The necessary assignments for fault f_1 (shown within ellipses in Figure 22) are 1 at line a (fault excitation), and 0 at line b and

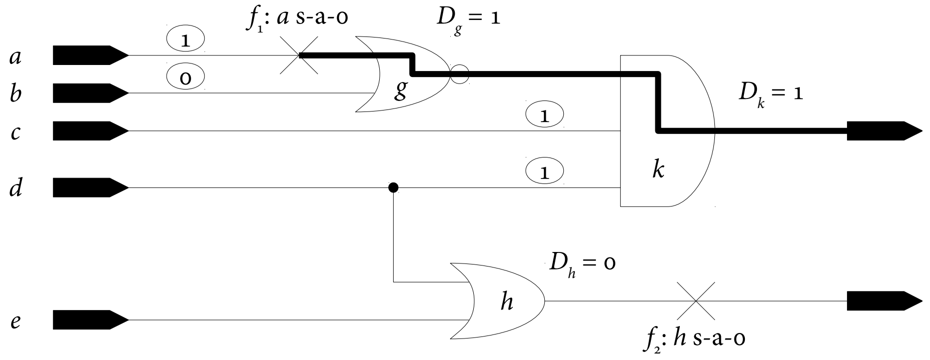


Figure 22. SAT-ATPG with dynamic compaction — extraction of necessary assignments

1 at lines c and d (fault propagation). Every test pattern p' that sets these four lines to these values is guaranteed to detect f_1, even if there are primary inputs that are assigned contradictory values by p and p'. Hence, the extraction of necessary assignments from internal lines gives the ATPG algorithm more flexibility while still guaranteeing the detection of all involved faults. In particular, if the fault effect has been propagated through multiple paths, then only the path with the smallest number of satisfied D-variables is considered for assignment extraction, such as to avoid an over-constraining of the test generation problem for subsequent faults. Finally, note that all D-variables assigned to 1, as well as all extracted assignments are contained in the Boolean solution that has already been computed by the GENERATE-TEST-PATTERN-procedure. Hence, no further simulation or complex analysis is necessary to extract the assignments.

Going back to the main procedure in Algorithm 5, the extracted assignments are added to the fault group assignment set *FGA* which is empty in the beginning and every time that a new fault group is opened. In the next iteration, the next processed fault is extended (line 8) such that any test pattern that detects the extended fault detects also all faults in the currently open fault group, whose necessary assignments are collected in *FGA*. Technically, this is implemented by adding *FGA* to the aggressor set of the fault to be extended. For example, assume that fault $f_2 : h$ s-a-0 is processed after $f_1 : a$ s-a-0 (Figure 22). Then, f_2 is extended to $f_2' :$ if $[a = 1, b = 0, c = 1, d = 1]$ h s-a-0. This implementation benefits from the CMS@ support already present in TIGUAN. Therefore, the core GENERATE-TEST-PATTERN-procedure does not need to be modified in order to support dynamic compaction.

Regarding the extraction of excitation assignments, it must be remarked that if the considered fault has more than one CMS@ victim, then simply adding the assignment of v to the value $\neg b_v$ for each victim line v is not adequate. That would result in an over-constrained test generation problem that would classify instances in which not all victims can be excited simultaneously as undetectable. However, that would be a wrong classification because the excitation of one victim suffices to detect a fault if that victim's fault effect can be propagated, even if the other victims cannot be excited simultaneously. In consequence, for general CMS@ faults with multiple victims, the implementation needs to impose the extracted conditions for victim excitation in the faulty instead of in the fault-free case. This is easily implemented by defining a new protected type of aggressor that allows the imposition of values on the B-variable instead of on the G-variable assigned to the aggressor line.

The processing of the fault list terminates when no unclassified faults remain, i.e. when the fault list becomes empty. Then, the last value of p_{group} is merged into the final test set (line 29). Otherwise, the test pattern for the last fault group would be lost if the last processed fault has not closed the last fault group.

Finally, the size of P can be further reduced by applying reverse-order fault simulation (ROFS or FLROFS) to the final test set (line 30). The fault simulator integrated into Tiguan was extended to perform these types of simulation in pattern-parallel fashion. In the case of FLROFS, which requires the test generation process to record which test pattern was the first to detect which fault, these data are recorded by a run of in-order fault simulation at the end of the test generation process. Due to the addition of new test patterns through merging, which dynamically modifies the final test set every time that a fault group is closed, the data needed for FLROFS cannot be collected by Tiguan's main test generation procedure.

6.3 Experimental evaluation

The dynamic compaction procedure was integrated into Tiguan using MiraXT as SAT solving back-end, but no thread-parallel SAT solving was employed. All measurements were performed on a 2.3 GHz AMD Opteron 64-bit computer with 64 GB RAM, but the memory use was limited to 4 GB per Tiguan process, given that MiraXT is a 32-bit application.

Tests for stuck-at faults in the combinational cores of NXP circuits were generated using Tiguan in three different configurations. No timeout was imposed on SAT solving such as to enable the classification of all faults. In this experiment, the input fault list was constructed in the following way. First, all s-a-0 and all s-a-1 faults located at the circuit's primary outputs are added to the fault list. Then, the fault list construction algorithm traverses the circuit in reverse topological order,

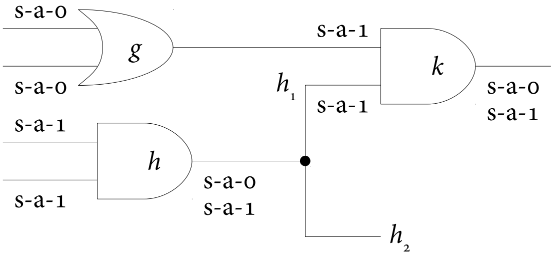


Figure 23. Construction of the fault list

i.e. starting at the primary outputs and ending at the primary inputs. For each gate *g*, the algorithm inserts all stuck-at faults located at any of *g*'s ports which are not equivalent to faults previously added to the fault list. For example, consider the circuit section shown in Figure 23. Assume that the s-a-0 and s-a-1 faults at the output of gate *k* have already been added to the fault list. Then, the algorithm adds only the s-a-1 faults located at the inputs of gate *k* to the fault list, because all s-a-0 faults located at any port of an AND gate are equivalent (all these faults require a test pattern that detects them to set all inputs of the gate to 1). Next, the algorithm considers gate *h*. Independently of which faults located on lines h_1 and h_2 have been added to the fault list, the algorithm adds both the s-a-0 and the s-a-1 fault at the output of gate *h* to the fault list, since stuck-at faults on a fan-out stem are not equivalent to faults on the fan-out branches. Then, the algorithm adds the s-a-1 faults located at the inputs of gate *h*. Finally, gate *g* is considered. No faults associated to *g*'s output port are added to the fault list, since these faults are equivalent to the faults associated to the first input of gate *k*, which has already been processed. Then, the algorithm only considers the s-a-0 faults at the inputs of gate *g*, since the s-a-1 faults are equivalent to the s-a-1 fault at the input of gate *k*, which is already in the fault list.

This algorithm renders a complete stuck-at fault list which is irredundant with respect to local fault equivalences [12], where the faults are ordered in reverse topological order with respect to their location. When not otherwise stated, this is the default fault list ordering employed in all experiments presented in this thesis and will be referred to as *reverse topological sorting* (RTOP) in Section 6.4, where different sorting strategies are evaluated. The experiment presented in this section uses reverse topological sorting.

Table 21 compares the pattern counts obtained by TIGUAN using the three different configurations. In the first configuration (column group *no compaction*), the generated test patterns were filled randomly, and 32-bit PPSFP (parallel-pattern single-fault propagation, see Section 2.6) fault simulation was used to drop faults detected by simulation. No static or dynamic compaction was used in this configuration. In the second configuration (column group *static compaction*), the generated patterns were relaxed using the input-output-cone analysis presented in Section 4.5, 32-bit fault dropping was performed using the relaxed patterns, and the final test set was compacted using a greedy merging algorithm: Given a sequence of test patterns $p_1, \ldots, p_r$, the algorithm tests the compatibility of each pattern p_i to each pattern p_j with $j > i$ and, if found compatible, p_i is replaced by $p_i \cap p_j$, and p_j is dropped from the test set. For reference, this configuration can be loosely regarded as a reimplementation of [68]. However, it must be noted that Tiguan does not perform the additional local analysis that is done in [68], and that [68] does not specify what static compaction algorithm was employed. A comparison with the numbers published in [68] is not presented. In preliminary experiments it was observed that Tiguan produced relatively large pattern counts for circuits for which [68] reported compact test sets, and vice versa. A possible reason for this phenomenon is that, although the same benchmark suite was used in [68], the circuit versions provided to the University of Bremen and to the University of Freiburg were probably synthesised using different options, thus resulting in different local structures, as indicated by the discrepancy in gate counts. For this reason, an objective comparison is not possible.

Finally, the last configuration (column group *dynamic compaction*) corresponds to the basic dynamic compaction procedure introduced in the previous section.

In order to further reduce the test set size, reverse-order fault simulation (ROFS) and forward-looking reverse-order fault simulation (FLROFS) were applied to the final test set in all three configurations. Hence, three numbers are quoted for each configuration. The first number (columns labelled *brofs*) is the size of the final test set before the application of reverse-order fault simulation, while the second number (columns labelled *arofs*) is the number of test patterns that remained after the application of ROFS. The third number (columns labelled *aflofs*) is the amount of test patterns that remained after the application of FLROFS instead of ROFS. In the second configuration, where static compaction was applied to the test set produced by the core test generation algorithm, also the pattern count before the application of static compaction is quoted (column *bcomp*).

Row *sum* quotes the total number of patterns that were generated for all circuits in each case, while row *norm sum 1* normalises these numbers with respect to the pattern count obtained without compaction and without reverse-order fault simulation (147,848). Row *norm sum 2* compares only the pattern counts obtained in combination with FLROFS.

Table 21

SAT-ATPG for stuck-at faults — static and dynamic compaction — test set size

	no compaction			static compaction				dynamic compaction		
circuit	brofs	arofs	aflrofs	bcomp	brofs	arofs	aflrofs	brofs	arofs	aflrofs
p35k	10,267	7,218	6,577	19,086	10,170	7,483	6,252	5,746	4,594	4,154
p45k	3,567	2,921	2,786	19,196	3,347	3,281	3,122	3,261	3,152	3,058
p77k	5,752	3,854	3,245	27,341	2,500	2,079	1,689	1,937	1,817	1,484
p78k	219	192	162	28,867	282	215	107	206	130	103
p81k	15,124	6,746	5,944	76,929	21,279	7,161	4,779	12,427	7,780	3,586
p89k	10,016	6,898	5,821	54,497	8,286	6,689	4,718	4,942	4,391	3,600
p100k	4,876	3,685	3,001	44,689	3,133	2,961	2,824	3,106	3,013	2,724
p141k	8,105	5,943	5,065	82,901	26,473	7,843	4,230	12,589	5,339	3,380
p267k	12,303	8,770	7,345	125,785	7,641	5,839	4,307	5,234	4,957	3,368
p269k	12,407	8,863	7,404	126,832	7,676	5,851	4,315	5,317	5,001	3,398
p286k	18,932	14,484	12,175	181,672	23,126	12,860	8,118	15,565	11,395	6,927
p295k	21,710	15,751	12,668	180,088	9,805	6,295	4,657	7,819	6,725	4,577
p330k	23,601	18,337	17,596	133,952	17,272	15,068	13,176	9,888	9,205	8,410
p378k	299	272	227	144,374	282	243	126	206	167	133
p469k	670	443	345	2,854	1,244	466	346	938	459	331
sum	147,848	104,377	90,361	1,249,063	142,516	84,334	62,766	89,181	68,125	49,233
norm sum 1	100%	70.6%	61.1%	844.8%	96.4%	57.0%	42.5%	60.3%	46.1%	33.3%
norm sum 2			100%				69.5%			54.5%

Circuit p388k was excluded from the table as the configuration using static compaction was not able to process it due to the memory limit per Tiguan process, which also prevented the processing of the three largest NXP circuits, p951k, p1522k and p2927k, using dynamic compaction.

In all three scenarios, ROFS and FLROFS leads to a large reduction of the pattern count, and FLROFS significantly outperforms ROFS in all cases.

In combination with FLROFS, dynamic compaction outperforms its static counterpart for every circuit with the exception of p378k. In total (row *norm sum 2*), static compaction improves the total pattern count by 30.5% with respect to the configuration without compaction, while dynamic compaction improves it by 45.5%, reducing the total pattern count by 21.6% with respect to static compaction.

It can also be observed that, using static compaction, FLROFS reduces the final amount of patterns by 55.9%, while the reduction is of only 44.7% using dynamic compaction. This means that FLROFS is less effective when applied after dynamic compaction than when applied after static compaction, which conversely shows that dynamic compaction is more effective in generating essential test patterns.

The only case in which both static and dynamic compaction were not able to reduce the test set size in combination with reverse-order fault simulation is circuit p45k. However, both static and dynamic compaction outperform the method without compaction when applied to this circuit without reverse-order fault simulation. The reason for this is that p45k is a circuit with very easy-to-solve stuck-at faults. Hence, the configuration that uses fault dropping based on randomly filled patterns is very effective in detecting faults by simulation. But the general observation is that the methods using compaction are superior to the method without compaction, and that employing FLROFS in combination with dynamic compaction results in the best pattern counts.

Table 22 lists the run-times in seconds needed by the three configurations in conjunction with FLROFS. The last two rows quote the accumulated run-time needed to process all circuits (row *sum*), and the total run-times normalised with respect to the time consumed by the first configuration (row *norm sum*).

In the first (column group *no compaction*) and third configurations (column group *dynamic compaction*), the total run-time (columns labelled *total*) comprises the time needed for the main test generation process (columns labelled *tpg*) and the time needed to apply forward-looking reverse-order fault simulation to the final test set (columns labelled *flrofs*). Hence, the numbers quoted in the *total*-columns are the sum of the numbers quoted in the two predecessor columns. In the second configuration (column group *static compaction*), where the static compaction algorithm is applied to the generated test set prior to FLROFS, also the time of the static compaction algorithm needs to be considered (column *comp*).

It is apparent that the run-time of the core test generation algorithm increases when static and dynamic compaction are applied. The reason for the difference between the first configuration (no compaction) and the second configuration (static compaction) stems from the difference in the number of test generation runs. The first configuration performs fault dropping using randomly specified test patterns, while the second configuration drops faults based on patterns that have been relaxed for better static compaction. Hence, in the first configuration, the amount of faults detected by simulation is significantly larger than the amount of faults detected by simulation in the second configuration, as shown by the immense difference in the number of test patterns produced by the core test generation algorithm in both cases (second versus fifth columns of Table 21). Hence, substantially fewer faults need to be targeted explicitly by the test generation process in the first configuration. The fact that the second configuration produces 8.4 times more patterns than the first, while the time needed for test generation grows by a factor of less than 2, implies that the average test generation time per fault is smaller when static compaction is

Table 22

SAT-ATPG for stuck-at faults — static and dynamic compaction — run-time

	no compaction (s)			static compaction (s)				dynamic compaction (s)		
circuit	tpg	flrofs	total	tpg	comp	flrofs	total	tpg	flrofs	total
p35k	1,219	259	1,478	1,429	677	279	2,385	1,889	151	2,040
p45k	57	64	121	157	110	58	325	222	61	283
p77k	5,771	839	6,610	6,123	268	370	6,761	6,306	262	6,568
p78k	5	31	36	284	64	13	361	283	12	295
p81k	243	583	826	1,278	10,598	586	12,462	1,460	401	1,861
p89k	209	542	751	677	2,397	381	3,455	772	246	1,018
p100k	112	286	398	599	384	166	1,149	689	178	867
p141k	1,324	590	1,914	6,255	10,502	1,295	18,052	7,502	699	8,201
p267k	522	1,492	2,014	3,999	7,268	1,073	12,340	4,053	586	4,639
p269k	478	1,200	1,678	5,146	7,506	1,078	13,730	4,279	703	4,982
p286k	4,114	3,512	7,626	13,837	38,231	5,660	57,728	16,730	3,331	20,061
p295k	1,288	3,176	4,464	8,427	34,220	1,260	43,907	11,149	1,981	13,130
p330k	15,007	6,661	21,668	20,915	16,061	3,116	40,092	28,535	1,606	30,141
p378k	31	183	214	5,596	1,166	59	6,821	4,226	79	4,305
p469k	33,437	1,620	35,057	26,883	13	800	27,696	129,652	1,370	131,022
sum	63,817	21,038	84,855	101,605	129,465	16,194	247,264	217,747	11,666	229,413
norm sum			100%				291%			270%

applied. However, this is due to the fact that most faults detected by simulation are very easy-to-detect ATPG instances. Therefore, in the first configuration, the test generation algorithm was applied to a larger fraction of hard-to-detect faults.

Regarding the run-time of the core test generation algorithm in combination with dynamic compaction (ninth column of Table 22), various factors have to be taken into consideration. On the one hand, the average SAT solving time increases when dynamic compaction is performed. The reason for this is that the processing of each fault becomes harder with growing size of the processed fault group. Since each new fault that is added to a fault group is passed the fault group's detection conditions in form of additional CMS@ aggressors, very large fault groups lead to extended faults that can be on average substantially harder-to-detect than the original faults. This is particularly visible for circuits with a large number of hard-to-detect faults, like p469k, where the test generation run-time increases substantially (129,652 seconds instead of only 26,883 seconds needed in the second configuration). On the other hand, since the dynamic compaction algorithm generates test patterns such as to target multiple faults, the likelihood that additional faults are detected by simulation increases, which means that less faults need to be targeted explicitly by the test generation process in comparison to the method that uses static compaction.

In summary, the run-time performance of the core test generation algorithm in combination with dynamic compaction depends on how long the fault groups become on average, on how hard the explicitly targeted faults are and on the efficacy of fault dropping which depends on the circuit's depth and number of inputs among other factors. Hence, the difference in time needed by the core test generation process in the second and in the third configuration varies strongly depending on the circuit, but in general this time is higher when dynamic compaction is employed. However, in the second configuration also the time needed to apply the static compaction algorithm to the generated test set has to be taken into account. This time (column *comp*) depends only on the number of test patterns that are processed (column *bcomp* in Table 21), and on the length of the test patterns, i.e. on the circuit's number of primary inputs, but not on the circuit's inherent hardness because the gate-level net list is not an input of the static compaction algorithm. This is perfectly illustrated by the fact that the smallest compaction time was measured for circuit p469k, which is the hardest benchmark from the point of view of test generation. Since the implemented static compaction has a quadratic run-time complexity in the worst case, the compaction time is higher than the test generation time in several cases. With the exception of circuit p469k, the sum of the these two times (columns *tpg* and *comp*) is consistently higher than the test generation time of the method that employs dynamic compaction.

Finally, the time needed to perform FLROFS in all three configurations has to be considered. Despite the efficiency of the used fault simulator, this time is not negligible due to the large number of simulated patterns. In fact, it constitutes 24.8% of the total run-time in the first configuration. However, as it depends roughly linearly on the number of test patterns that need to be simulated (columns labelled *brofs* in Table 21), it decreases sensibly in the configurations that use static and dynamic compaction as less patterns are simulated in these cases.

Given all these factors, it is hard to predict the exact difference between the total run-time needed to process each circuit using the methods with static and dynamic compaction. In fact, there are circuits for which static compaction requires considerably more time than dynamic compaction (e.g. p81k, where static compaction needs 6.7 times more time than dynamic compaction) as well as circuits for which the opposite is the case (e.g. circuit p469k, where dynamic compaction needs 4.7 times more time than static compaction). However, the method using dynamic compaction was able to process all circuits in 7.2% less time than the method using static compaction. Disregarding the exceptionally hard circuit p469k, the dynamic-compaction configuration processed all circuits in 55.2% less time than the static-compaction configuration. That makes it evident that the proposed dynamic compaction method is superior to static compaction in regard to both pattern counts and run-time.

6.4 Enhanced dynamic compaction

The basic dynamic compaction algorithm presented in the previous section was extended such as to enable the formation of longer fault groups. The modified procedure, which is described in Algorithm 6, is controlled by a new integer parameter $\alpha > 0$, which is called the *incompatibility conflict limit*. This parameter determines how many faults have to lead to an incompatibility conflict with the currently open fault group before the fault group is closed. In order to manage the data that depend on this new parameter, two new variables *cc* and *IF* are defined. The former is a counter that will hold the current number of incompatibility conflicts that have arisen for the current fault group, while the latter will hold the faults that need to be reprocessed after they have been found to be incompatible to the currently open fault group. These two variables are initialised to 0 and $\varnothing$, respectively, at the end of the initialisation block (lines 3–7).

Like the basic algorithm from the previous section, the ATPG procedure iterates over the fault list. In each iteration (lines 8–26), the algorithm selects the first fault f from the fault list F, but in contrast to the first algorithm, the fault is removed immediately from F (line 10). Then, f is extended by the fault group assignments of the current group (line 11), and the test generation procedure attempts to find a test pattern that detects the extended fault f' (line 12).

Then, the enhanced algorithm distinguishes the same three cases handled by the basic algorithm. In the first case (lines 13–14), f has been processed independently of any fault group, while the test generation process has not been able to generate a test pattern for f. Thus, f is classified as undetectable or aborted, and automatically excluded from further processing, as it has already been deleted from F.

In the second case (lines 15–22), the test generation for the extended fault f' has failed, which shows that f is incompatible to the currently open fault group. Hence, the conflict counter *cc* needs to be incremented, and f is added to *IF* which holds the faults that need to be reprocessed. If the conflict limit has been reached ($cc = \alpha$), the current fault group is closed, i.e. the current value of p_{group} is merged into the final test set, fault dropping is performed, and the variables p_{group}, *FGA* and *cc* are reinitialised. The close-current-fault-group-procedure in line 19 is equivalent to lines 17–21 in Algorithm 5. Then, all faults that have been collected for reprocessing are moved back from the list of incompatible faults *IF* to the main

Algorithm 6
Enhanced dynamic compaction for SAT-ATPG

```
Inputs: CMS@ fault list $F$, integer conflict limit $\alpha > 0$
Output: compact test set $P$
ATPG-WITH-ENHANCED-DYNAMIC-COMPACTION($F, \alpha$) {
    sort $F$
    $P := \varnothing$                                   ▷ initialisation
    $p_{\text{group}} := nil$
    $FGA := \varnothing$
    $cc := 0$                                            ▷ conflict counter
    $IF := \varnothing$                                  ▷ holder of incompatible faults
    while $F$ is not empty do {
        $f$ := first fault in $F$
        remove $f$ from $F$
        $f' :=$ EXTEND-FAULT-DESCRIPTION($f, FGA$)
        $p :=$ GENERATE-TEST-PATTERN($f'$)
        if $p = nil$ and $p_{\text{group}} = nil$ then {         ▷ $f$ is undetectable or aborted
            classify $f$ appropriately
        } else if $p = nil$ and $p_{\text{group}} \neq nil$ then {  ▷ $f$ is incompatible
            increment $cc$ by 1
            add $f$ to $IF$
            if $cc = \alpha$ then {                      ▷ conflict limit reached
                CLOSE-CURRENT-FAULT-GROUP( )
                $cc := 0$
                move all faults in $IF$ to the beginning of $F$
            }
        } else {                  ▷ current fault group successfully enlarged by $f$
            ENLARGE-CURRENT-FAULT-GROUP($f$)
        }
    }
    MERGE($P, p_{\text{group}}$)
    ROFS-OR-FLROFS($P$)
    return $P$
}
```

fault list *F*. In order to preserve the benefits of the sorting performed on the input fault list at the beginning of the procedure, *IF* is implemented as a stack, and the faults in *IF* are moved to the beginning of *F* such that they occupy the same position with respect to the ordering of *F* after the initial sorting.

If the conflict limit has not been reached yet, the fault group is left open and the procedure attempts to add the next processed fault to the fault group.

In the last case (lines 23–24), where the test generation procedure has been successful, the fault group is enlarged like in the basic algorithm. The ENLARGE-CURRENT-FAULT-GROUP-procedure in line 24 is equivalent to lines 22–26 in Algorithm 5. The rest of the procedure (lines 27–29) is left unmodified.

A series of experiments were performed in order to examine how the dynamic compaction procedure's performance is influenced by the conflict limit α in combination with different sorting strategies. All measurements were performed on a 2.3 GHz AMD Opteron 64-bit computer with 64 GB RAM, but with a memory limit of 4 GB per TIGUAN process. FLROFS was employed in all experiments.

In the first experiment, tests for stuck-at faults were generated for larger ISCAS'85 circuits and for the combinational cores of larger ISCAS'89 circuits without imposing a time limit on SAT solving. For each circuit, five different α-values ($\alpha = 1$, which corresponds to the basic algorithm evaluated in the previous section, and $\alpha = 25, 50, 75, 100$) were tested in combination with three fault list sorting strategies, hence resulting in fifteen different configurations. Several simple sorting strategies were implemented, in part based on sorting techniques proposed in [185]. In this experiment, only the three strategies that lead to the best results in preliminary experiments were considered:

- *Reverse topological sorting* (RTOP) — the fault list contains all stuck-at faults up to local fault equivalence, where the faults are processed in reverse topological order with respect to their location. This is the default sorting used by TIGUAN in all experiments shown in this thesis (see Section 6.3).
- *Topological sorting* (TOP) — the fault list contains the same faults as in RTOP, but the faults are processed in topological order with respect to their location, i.e. faults located at the primary inputs are processed first, and faults located at the primary outputs are processed last.
- *FFR-based sorting* (FFR) — the fault list contains the same faults as in RTOP, but the faults are sorted based on the fan-out-free region to which the fault location belongs. First, the circuit is partitioned into FFRs, and the faults in the input fault list are assigned to FFRs depending on their location. Then, the FFRs are sorted in falling order with respect to their size, i.e. to the number

of faults that have been assigned to them. If two FFRs have the same size, the FFR whose root gate is closer to the primary inputs of the circuit is given precedence.

Let $\mathfrak{G}_1, \dots, \mathfrak{G}_n$ be the ordered FFRs of a circuit, with $\mathfrak{G}_1$ being the largest and $\mathfrak{G}_n$ being the smallest FFR. For $i = 1, \dots, n$, let $F_i := f_1^i, \dots, f_{m_i}^i$ be the sequence of faults located within $\mathfrak{G}_i$ sorted in reverse topological order, i.e. f_1^i and f_2^i are the s-a-0 and s-a-1 faults located at the output of $\mathfrak{G}_i$'s root gate. Then, the fault list is constructed by iterating over the FFRs. In each iteration, the first fault of each sequence F_i is added to the final fault list and removed from F_i. If a sequence becomes empty, it is excluded from the next iteration. The procedure terminates when all sequences have become empty. Then, the final fault list has the form $f_1^1, f_1^2, \dots, f_1^n, f_2^1, f_2^2, \dots, f_2^n, f_3^1, f_3^2, \dots, f_3^n, \dots$.

The intuition behind this sorting strategy is that the test generation for faults located within one FFR is more likely to result in conflicting justification conditions than the test generation for faults located within different FFRs. Hence, it is expected that processing faults located in different FFRs after each other will lead to fewer fault group incompatibilities. Also, by processing first the faults located on the root gates of FFRs, it is expected that more faults can be detected by simulation.

The total number of patterns generated for all circuits (after the application of FLROFS) and the total run-time needed to process all circuits are shown in graphical form in Figure 24. The abscissa corresponds to the different values of α, while the pattern counts (left-hand-side ordinate scale) and the run-times in seconds (right-hand-side ordinate scale) are shown in the form of six curves — one test size and one run-time curve for each sorting strategy.

The first observation is that increasing the conflict limit α succeeds in reducing the pattern counts, where the largest difference can be observed when α is incremented from 1 to 25. Although incrementing α to higher values consistently results in lower pattern counts, the difference tends to become smaller for each increment of α by 25. An interesting observation is that the same progression can be observed for all three sorting strategies. Hence, the parameter α influences the amount of generated patterns in a form that is independent of the used sorting strategy. Although the differences between the different sorting strategies are small, topological sorting results in the lowest pattern counts for all α-values.

In experiments conducted to examine the impact of fault dropping both with and without dynamic compaction, it was observed that each run of the fault dropping procedure tends to detect fewer faults by simulation than the previous run, espe-

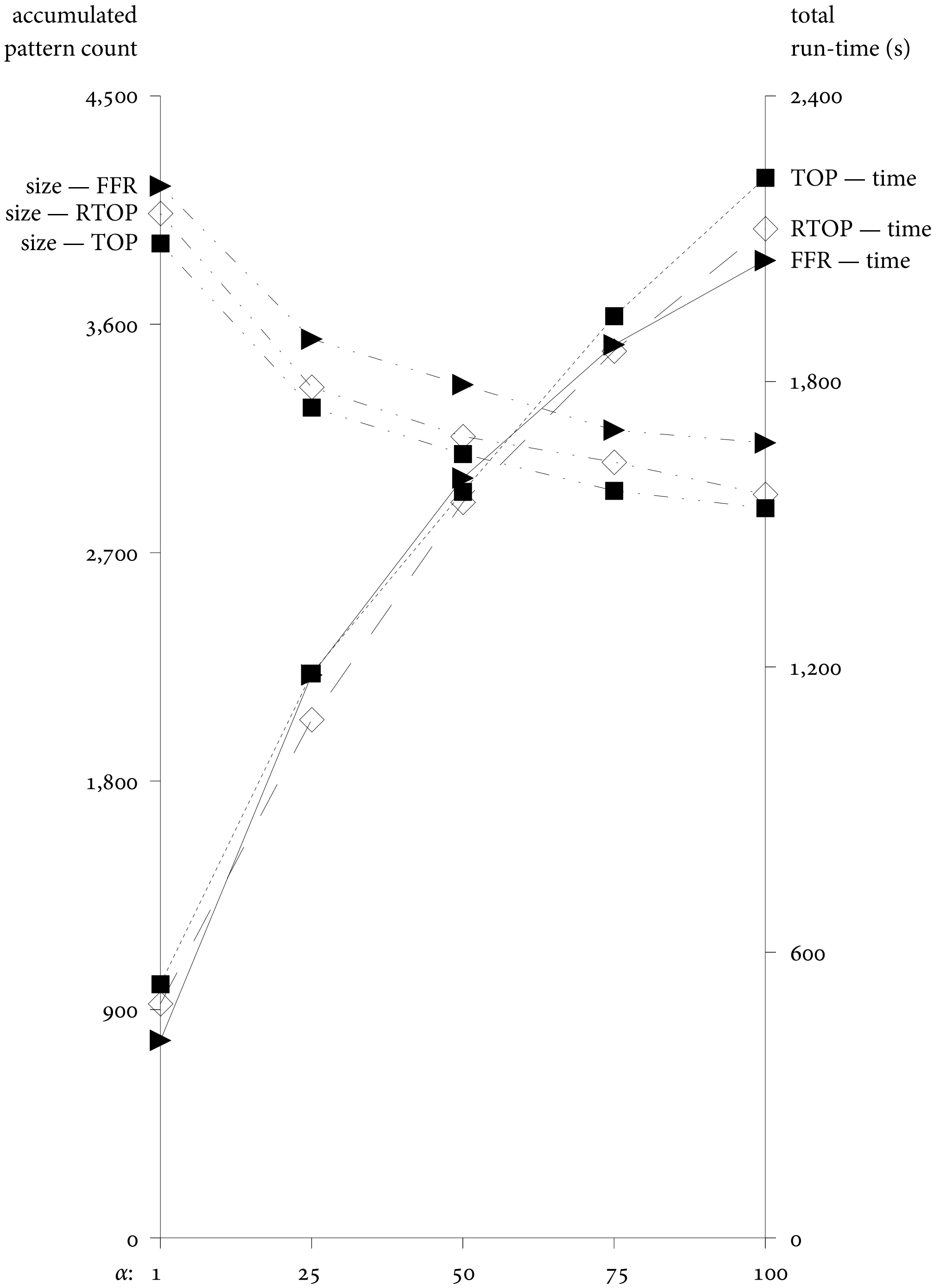


FIGURE 24. ENHANCED DYNAMIC COMPACTION FOR SAT-ATPG — IMPACT OF CONFLICT LIMIT α AND FAULT LIST SORTING

cially in conjunction with topological fault list sorting. Consequently, the capacity that the test patterns generated at the beginning of the process have to detect a large number of faults can be decisive in obtaining compact test sets. This is a possible explanation for the good performance of the TOP sorting strategy regarding pattern counts in this experiment. Using this strategy, the first fault groups that are constructed are composed of faults with very small input cones. Hence, the first fault groups are likely to become very large due to a reduced probability of fault group incompatibility. Therefore, the first fault group patterns that are added to the test set are likely to detect more faults by simulation. However, larger fault groups lead to an increased average test generation time per fault. In consequence, the TOP strategy has the largest run-times, especially for $\alpha = 100$, where the constructed fault groups are allowed to become particularly large.

In contrast, the FFR-based sorting strategy results in slightly higher pattern counts for all α-values, but it achieves better run-times, especially for $\alpha = 1$ and $\alpha = 100$, which indicates that combining faults from different FFRs into the same fault group resulted in extended ATPG instances that were easier to detect due to the reduced probability of conflict at the time of propagating fault effects towards primary outputs. But, since the largest FFRs were given precedence, the input cones of the faults that were collected in the first fault groups were larger. Thus, the test patterns that were constructed at the beginning of the process were less likely to detect as many faults by simulation as in the case where the TOP strategy was used. In sum, this shows that it is very hard to optimise pattern counts and run-times using one single strategy.

In contrast to the progression of the pattern-count curves, the total run-times measured for different α-values progress slightly differently for each sorting strategy. It is difficult to assess the exact reasons for the specific progression of each curve, given that the factors that influence the run-time of the dynamic compaction procedure are multiple and not mutually independent. But, in general, it can be observed that incrementing α leads to run-times that grow faster than the pattern counts fall. Still, the observed run-times grow only roughly linearly in α.

In summary, although the run-time increment incurred by using large α-values is not critical, the best compromise between pattern count reduction and run-time increment is achieved for α-values of 25 and 50. Also, given the small differences in run-time between the different sorting strategies, the TOP strategy, which achieved the most compact test sets in all cases, can be regarded as the best strategy for SAT-ATPG with dynamic compaction.

The detailed results obtained in this experiment using the TOP strategy are quoted in Table 23. The row labelled *sum* shows the total number of patterns generated

Table 23

Enhanced dynamic compaction for SAT-ATPG with topological fault list sorting — impact of conflict limit α — ISCAS circuits

	test set size					run-time (s)				
circuit	$\alpha = 1$	$\alpha = 25$	$\alpha = 50$	$\alpha = 75$	$\alpha = 100$	$\alpha = 1$	$\alpha = 25$	$\alpha = 50$	$\alpha = 75$	$\alpha = 100$
c1355	88	85	85	85	85	3.6	10.2	17.5	22.6	25.9
c1908	125	116	115	122	124	2.8	13.4	19.8	29.4	31.2
c2670	124	92	84	72	70	7.0	15.4	22.7	26.1	32.9
c3540	172	140	130	126	124	10.4	39.9	63.0	88.6	101.0
c5315	123	93	86	84	85	10.0	30.9	42.1	52.9	56.3
c6288	53	41	36	32	36	24.6	57.0	83.6	108.4	138.8
c7552	241	174	141	129	120	17.5	54.3	78.7	106.5	112.7
cs00820	108	104	107	101	104	0.7	3.1	4.7	5.6	7.3
cs00832	107	105	102	103	102	0.7	2.9	3.8	6.4	7.7
cs00838	163	152	150	150	149	1.4	8.0	10.1	15.7	19.6
cs00953	86	86	85	86	86	0.7	3.1	5.1	6.3	8.9
cs01196	163	134	134	130	132	1.5	7.2	11.4	15.3	17.5
cs01238	161	142	142	133	142	1.8	7.9	13.2	15.5	17.6
cs01423	61	46	44	42	39	1.9	4.6	6.7	9.4	10.8
cs01488	124	117	114	112	116	0.8	4.0	5.1	7.6	10.5
cs01494	127	119	113	112	107	1.0	4.2	6.3	7.6	9.4
cs05378	199	159	134	134	136	5.5	23.2	32.4	42.1	46.8
cs09234	347	238	206	196	182	19.1	61.8	98.1	114.5	141.0
cs13207	302	281	279	261	247	23.4	104.8	156.1	205.3	253.0
cs15850	298	208	197	183	178	40.0	117.4	163.0	237.9	283.1
cs35932	36	35	37	31	29	187.5	259.3	284.3	328.0	277.8
cs38417	291	236	230	215	193	94.0	179.6	222.5	246.0	300.5
cs38584	420	369	338	305	289	77.2	173.9	217.6	238.8	317.5
sum	3,919	3,272	3,089	2,944	2,875	533.1	1,186.2	1,568.1	1,936.8	2,227.7
norm sum	100%	83.5%	78.8%	75.1%	73.4%	100%	223%	294%	363%	418%

for all circuits and the total run-time required to process all circuits, and the row labelled *norm sum* shows these numbers normalised with respect to the values obtained for $\alpha = 1$, which corresponds to the basic dynamic compaction algorithm presented in Section 6.2.

Regarding each circuit separately, it can be seen that increasing the value of α does not necessarily have the same effect on all circuits. For example, there are circuits (e.g. cs00953) for which modifying the value of α had nearly no effect. Also, there are circuits (e.g. c1908) for which setting $\alpha = 75$ and $\alpha = 100$ results in slightly higher pattern counts than $\alpha = 50$, thus confirming that, although the average length of the constructed fault groups has a large influence on the final test size, the performance of the dynamic compaction method depends on several factors. At the same time, there are circuits (e.g. cs09234 and cs38584) for which the difference between α = 75 and α = 100 is still significant. In contrast, increasing α-values results in consistently larger run-times for all circuits. This corroborates the observation that, in general, α-values below 50 lead to the best compromise between pattern counts and run-time.

After this observation, the method was applied in combination with the TOP sorting strategy and FLROFS to NXP circuits using three different α-values: α = 1, which corresponds to the basic dynamic compaction algorithm presented in Section 6.2, and α = 25 and α = 50. A SAT solving timeout of 1,000 seconds per SAT instance was used in order to prevent too long fault groups from resulting in an explosion of the total run-time.

The results are summarised in Table 24 (a). The first column group (*test set size*) quotes the number of generated test patterns that remained after the application of FLROFS. Due to the SAT solving timeout of 1,000 seconds, not all faults were classified in this experiment. Although it is safe to assume, based on previous experimental evidence, that most aborted faults are undetectable, from the point of view of compaction the worst case would occur if all aborted faults were detectable by only one test pattern each, and if that test pattern could not be merged with the other generated test patterns. Hence, in the worst case, all faults in the circuit can be tested using a maximum number of patterns that equals the size of the generated test set plus the number of aborted faults. This number is quoted in the second column group (*worst-case test set size*). The last column group (*run-time*) quotes the total run-time in seconds.

As can be seen, increasing the parameter α is effective in reducing pattern counts also when the algorithm is applied to large industrial circuits. Both the actual number of generated test patterns and the worst-case number of patterns are reduced by roughly 25% if α is increased from 1 to 25, at the expense of a run-time increment by a factor of 2.6. Therefore, using α = 25 can be regarded as computationally feasible. Although incrementing the value of α to 50 results in even better pattern counts, the difference in pattern counts between α = 25 and α = 50 is smaller. However, in comparison to the smaller circuits, it can be seen that larger α-values have a larger influence on run-time efficiency.

The results obtained for circuit p469k are shown separately, including more details, in Table 24 (b). This circuit is known for leading to extremely hard SAT instances despite its relatively small size. The second and third columns of the table quote the actual test set size after the application of FLROFS and the worst-case test size

Table 24

Enhanced dynamic compaction for SAT-ATPG with topological fault list sorting — impact of conflict limit α — NXP circuits

(a) all circuits without p469k

	test set size			worst-case test set size			run-time (s)		
circuit	$\alpha = 1$	$\alpha = 25$	$\alpha = 50$	$\alpha = 1$	$\alpha = 25$	$\alpha = 50$	$\alpha = 1$	$\alpha = 25$	$\alpha = 50$
p35k	4,037	2,715	2,212	4,273	2,946	2,461	2,216	8,849	13,653
p45k	3,095	2,563	2,387	3,095	2,565	2,390	318	1,461	2,495
p77k	1,349	1,133	1,070	1,366	1,158	1,115	6,767	10,583	14,592
p78k	104	83	84	104	83	84	1,134	2,998	4,478
p81k	1,553	1,121	997	1,564	1,133	1,005	1,677	3,594	5,645
p89k	3,925	2,647	2,135	3,927	2,670	2,162	1,440	3,177	4,374
p100k	2,757	2,383	2,281	2,763	2,399	2,293	1,090	3,345	5,263
p141k	3,385	2,234	1,932	3,535	3,106	2,992	9,805	22,127	27,607
p267k	2,859	2,139	2,026	2,868	2,166	2,064	5,872	15,213	23,070
p269k	2,901	2,155	2,062	2,911	2,189	2,108	5,930	14,681	23,289
p286k	6,424	4,823	4,455	6,532	5,041	4,756	21,480	54,222	69,370
p295k	5,971	5,359	5,116	5,983	5,389	5,151	12,923	38,273	52,871
p330k	6,493	4,205	3,871	6,517	4,258	3,951	33,384	118,249	182,311
p378k	129	106	103	129	106	104	38,274	100,769	114,095
p388k	2,821	1,886	1,720	2,873	2,032	1,893	20,659	33,151	34,937
sum	47,803	35,552	32,451	48,440	37,241	34,529	162,969	430,692	578,050
norm sum	100%	74.4%	67.9%	100%	76.9%	71.3%	100%	264%	355%

(b) detailed results for circuit p469k

			average per SAT instance				
	test set size				run-time (s)		
	actual	worst case	clauses	variables	formulation	solving	total time (s)
$\alpha = 1$	333	338	202,801	53,674	0.112	24.5	165,246
$\alpha = 25$	319	324	269,297	69,931	0.123	98.0	2,037,428
$\alpha = 50$	308	313	277,670	71,956	0.123	106.9	3,554,034

which is computed in the same way as for the circuits listed in Table 24 (a). As an indicator of the average SAT formula size, the next two columns quote the number of clauses and Boolean variables contained on average in each generated formula. Note that this includes all generated formulae, i.e. also the formulae that represent the extended CMS@ faults. Consequently, these numbers are increased by roughly 30% for $\alpha = 25$, because the fault groups are given the opportunity to become longer. Although these numbers grow again when α is incremented to 50, the difference is not as significant as for the α-increase from 1 to 25. This means, that the fault groups do not become significantly longer for $\alpha = 50$.

The next two columns list the average time in seconds needed to formulate one SAT formula (column *formulation*) and the average time to solve it (column *solving*). Since larger fault groups mean that the modelled circuit area becomes larger as well, the SAT formulation time also increases slightly for $\alpha = 25$ and $\alpha = 50$, but the increment is minimal. However, the average SAT solving time is quadrupled when α is incremented from 1 to 25. But it grows only slightly when α is further incremented to 50, which agrees with the observation that incrementing α to 50 does not contribute significantly to fault group elongation.

In contrast, α has a large impact on the total run-time (last column), which increases by a factor of nearly 20 when α is incremented from 1 to 25, and nearly doubles again when α becomes 50. The reason why α has a larger influence on the total run-time than on the average SAT solving time is the following: For any fixed α-value, each created fault group is closed only after α incompatibilities have been found, i.e. after the unsatisfiability of α SAT formulae has been determined. Thus, for $\alpha = 25$, at least 25 times more unsatisfiable SAT instances had to be solved than for $\alpha = 1$, and the same increment applies when α is further increased to 50. The results obtained for all other circuits (Tables 23 and 24 (a)) show that this is in general not an impediment for the successful application of the enhanced dynamic compaction procedure, even for α-values as large as 100. p469k constitutes an extreme example, given that it leads to a large number of unsatisfiable instances, and that its unsatisfiable instances are significantly harder to solve in comparison to its satisfiable instances. Hence, the implementation of an adaptive mechanism that prevents extreme behaviour on exceptionally hard circuits is an important direction for future research.

Disregarding p469k, the experiments demonstrate the potential of the basic dynamic compaction procedure presented in this chapter. By changing only one parameter, it has been possible to significantly improve the pattern counts with feasible computational effort, for both ISCAS and large industrial circuits.

For reference, Table 25 shows a comparison between the best pattern counts achieved by TIGUAN using the proposed dynamic compaction procedure, and the number of patterns produced by a commercial ATPG tool that implements a structural algorithm. The results quoted for the commercial tool were obtained in "high compaction effort"-mode.

Tables 25 (a) and 25 (b) list the results obtained for ISCAS'85 and ISCAS'89 circuits, respectively. Neither TIGUAN nor the structural tool aborted the processing of

Table 25

Dynamic compaction for SAT-ATPG — best Tiguan pattern counts from Tables 23 and 24 in comparison to a commercial tool (structural ATPG)

(a) iscas'85 circuits

circuit	Tiguan	structural
c1355	85	95
c1908	115	140
c2670	70	81
c3540	124	143
c5315	85	101
c6288	32	33
c7552	120	125
sum	631	718

(b) iscas'89 circuits

circuit	Tiguan	structural
cs00820	101	121
cs00832	102	119
cs00838	149	155
cs00953	85	97
cs01196	130	159
cs01238	133	163
cs01423	39	66
cs01488	112	128
cs01494	107	134
cs05378	134	136
cs09234	182	182
cs13207	247	281
cs15850	178	154
cs35932	29	51
cs38417	193	135
cs38584	289	167
sum	2,210	2,248

(c) nxp circuits

	Tiguan		structural ATPG		
circuit	no compaction	compaction (wc)	test size	aborts	wc test size
p35k	10,267	2,461	1,542	–	1,542
p45k	3,567	2,390	2,099	14	2,113
p77k	5,752	1,115	496	3,575	4,071
p78k	219	83	69	–	69
p81k	15,124	1,005	336	201	537
p89k	10,016	2,162	719	1	720
p100k	4,876	2,293	2,050	251	2,301
p141k	8,105	2,992	601	21	622
p267k	12,303	2,064	899	–	899
p269k	12,407	2,108	887	10	897
p286k	18,932	4,756	1,192	56	1,248
p295k	21,710	5,151	1,688	20	1,708
p330k	23,601	3,951	2,493	173	2,666
p378k	299	104	75	–	75
p388k	11,756	1,893	550	70	620
p469k	670	313	390	221	611
sum	159,604	34,841	16,086	4,613	20,699

any faults. The numbers quoted for Tiguan correspond to the lowest pattern count quoted in columns 2–6 of Table 23. It can be observed that the proposed compaction method enabled Tiguan to test all faults in iscas circuits using less patterns than the commercial, structural tool.

Table 25 (c) quotes the results obtained for nxp circuits. Given the large number of aborts produced by the structural tool for some circuits (e.g. p77k), an exact comparison is not possible. Nevertheless, this table provides a reference for the advancement of SAT-ATPG with respect to test compactness that is made possible by the proposed dynamic procedure. The first column group quotes pattern counts achieved by Tiguan. The column labelled *no compaction* corresponds to the second column of Table 21, which represents the number of patterns produced by Tiguan using 32-bit fault dropping, which is very effective in detecting a large number of faults by simulation, but without any additional compaction effort. The column labelled *compaction (wc)* corresponds to the lowest pattern count quoted in columns 5–7 of Table 24 (a), i.e. to the best worst-case pattern count achieved by Tiguan employing enhanced dynamic compaction and topological fault list sorting. The last three columns of the table quote the number of patterns the structural tool generated with high compaction effort, the number of aborted faults and the worst-case pattern count which is the sum of the two previous numbers. Although Tiguan's worst-case pattern counts are still higher than those of its structural counterpart, it is evident that the proposed dynamic compaction procedure allowed Tiguan to reduce the total pattern count for all circuits from 7.7 times the worst-case pattern count of the structural tool to only 1.7 times that number with feasible computational effort.

6.5 Conclusions

This chapter presented a dynamic compaction procedure for SAT-based ATPG which scales to large industrial designs and outperforms the static compaction method both in terms of test compactness and run-time. In fact, the dynamic compaction procedure enabled the SAT-based tool Tiguan to generate more compact test sets for iscas circuits than a commercial ATPG tool that implements a structural algorithm. Regarding the application to the large industrial circuits provided by NXP, the test sets produced by the commercial tool are still more compact than those produced by the SAT-based method, but the dynamic compaction procedure allowed Tiguan to significantly shrink the gap between structural and SAT-based ATPG in terms of test compactness.

The most important property of the presented technique is its tight integration into the ATPG tool. The technique makes use of the SAT-ATPG core's specific data structures and interfaces. This allows the dynamic compaction procedure to impose the smallest possible number of necessary conditions on internal lines of the circuit when the enlargement of the currently open fault group is attempted. Hence, the test generation algorithm encounters a less constrained instance that is more likely and more efficiently solvable. Moreover, the extraction of necessary conditions from internal lines of the circuit is more suitable for combination with SAT-ATPG than the traditional extraction from assignments made to the primary inputs, given that SAT-based test generation produces only patterns that are fully specified within the modelled circuit section.

Although the implemented form of extraction of minimal assignments does not require the relaxation of the generated test patterns, an input-output-cone analysis is performed for this purpose in order to allow for the merging of new test patterns into the final test set. Since the relaxation is performed for every pattern that is generated, the local relaxation analysis presented in [68] was not performed on top of the input-output-cone analysis, such as to avoid affecting the overall run-time efficiency of the SAT-based test generation process.

Different SAT, QBF and simulation-based approaches for the generation of test patterns with a maximum amount of unspecified bits have been published recently [194, 212]. In [212], a QBF-based method is presented which is able to generate tests with a provably optimal fraction of unspecified bits. Optimal results were generated at the expense of impractically large run-times, but that allowed the authors of [212] to evaluate alternative methods that achieve nearly-optimal densities of unspecified bits. The incorporation of such techniques would allow for better merging of the patterns generated by the dynamic compaction procedure into the final test set.

An important topic for future research is the enhancement of the run-time efficiency of the proposed dynamic compaction technique. A promising approach consists in utilising incremental SAT solving to reduce the average SAT solving time, which grows strongly when fault groups become very large. Since every new SAT instance that is generated for a given fault group shares a large part of the circuit model covered by the previous SAT instance, it is expected that the application of incremental learning within each fault group will lead to strongly reduced SAT solving times. However, in contrast to the fault clustering technique discussed in Section 5.2, where all faults that will be processed using one single incremental SAT solving process are given advance, in the presented dynamic compaction method, the composition of fault groups is not known at first. Therefore, the approach of the fault clustering technique, which generates only one SAT formula and solves it multiple times using different sets of assumptions, is not directly applicable to fault groups in dynamic compaction. Instead, the underlying circuit model (i.e. the coloured parts of the circuit for a given fault and the assignment of Boolean variables to lines in those circuit sections) needs to be extended dynamically for each new fault that is tested in combination with a fault group. In this context, it is important to guarantee that all lines in the circuit are represented by the same Boolean variables in all SAT formulae that are processed incrementally. In sum, the dynamic extension of the underlying circuit model has to be performed such as to satisfy this condition and such that a minimum number of Boolean variables is used.

A further point of interest is the suitability of SAT-based ATPG in the context of test compression. State-of-the-art test compression methods [192] rely on very high densities of unspecified bits. Alternative approaches to test compression which utilise the strengths of SAT-based ATPG, such as the ability to set a large number of constraints and to quickly identify unsolvable instances, are required.

7

Complex fault models and optimisation problems

This chapter focuses on the use of SAT-based methods for the generation of test patterns for complex fault models. After a motivation for the need of complex fault models, this chapter introduces a further application of the CMS@FM that has not been discussed in previous chapters. Then, an extension of the CMS@FM is introduced — the ECMS@FM. Like its predecessor, the ECMS@FM allows the definition of multiple stuck-at victims, but it also enlarges the range of possible conditions that can be imposed on aggressor lines, and it also enables the specification of additional optimisation goals. The text explains the implementation of ECMS@-based SAT-ATPG, which was deployed into Tiguan utilising the SAT solver antom's ability to solve SAT problems with qualitative preferences. Finally, two important applications of the new ECMS@FM are discussed and evaluated.

Author's contribution — The author's contribution consisted in the definition of the new ECMS@ fault model, and the adaption of the existing SAT-ATPG framework in order to perform ECMS@-based ATPG. In addition, the author explored applications of the new model that would result in particularly hard ATPG instances and would thus demonstrate the applicability of SAT-ATPG to test generation problems that cannot be solved trivially using structural test generation algorithms.

Parts of the work covered in this chapter have been published in [J2, C16, C7, W2] (see author's publications on pages 223–226).

7.1 Introduction

As was discussed earlier, the SAFM has been the dominant fault model used in practical applications, especially because years of practice have shown that test patterns generated for stuck-at faults cover many permanent defects. However, it has been shown that the SAFM does not reflect accurately several defect types encountered in the currently dominant CMOS technology [91, 110, 161, 11]. Approaches like gate-exhaustive [169, 43] or n-detection [186, 38] testing, which attempt to overcome this shortcoming by testing each stuck-at fault many times under different conditions, increase the coverage of realistic defects but fail to address more specific failures. For instance, shorts and opens account for a large portion of physical defects in CMOS ICs [91, 49], and the SAFM provides only a very rough approximation to the behaviour caused by these defects, especially considering that a substantial fraction of shorts and opens are resistive [200]. Also phenomena that affect signal integrity, such as capacitive crosstalk [149, 42, 261, 143], ground bounce [236] and power supply noise [228, 229], belong to the type of complex defects that cannot be modelled properly using the SAFM.

As a consequence, sophisticated *non-standard fault models* have been defined. One example, the RBF model, was introduced in Section 2.5. However, such models can reach a complexity level that makes it necessary to redefine the concepts of detectability and fault coverage. Moreover, even the implementation of dedicated ATPG tools for each individual fault model is often required [77, 182, 118, 92]. Aside from the cost of implementing multiple tools, each tool may need to model the problem at its own abstraction level, which complicates the integration of different tools. For this reason, the need for more generic approaches to model complex faults has been recognised in academia [64, 119] and industry [144, 158]. For example, in [64], *fault tuples* are used to model arbitrary misbehaviour in cycle timing. [144] defines the *generalised fault model* as a means of description of crosstalk-related defects. In [119], a similar model is used for diagnosis without fault dictionaries. The recent publication of [158] corroborates that *user-defined fault models* have gained relevance also in industry.

The author of this thesis introduced the CMS@FM (see Section 4.2) in [54]. This fault model allowed the development of one single SAT-based ATPG framework able to handle the SAFM and also more complex approaches like gate-exhaustive testing without the need to modify the basic algorithm for each fault model. Instead, the original problem is mapped to a set of CMS@ faults that can be processed efficiently thanks to SAT-based techniques, which are especially suitable for the application to hard-to-detect and undetectable ATPG instances, as shown in Chapters 4 and

5. In this chapter, a further application of the CMS@FM is presented. A *sectioning analysis* is performed on resistive-bridging faults in order to perform CMS@-ATPG based on the extracted information.

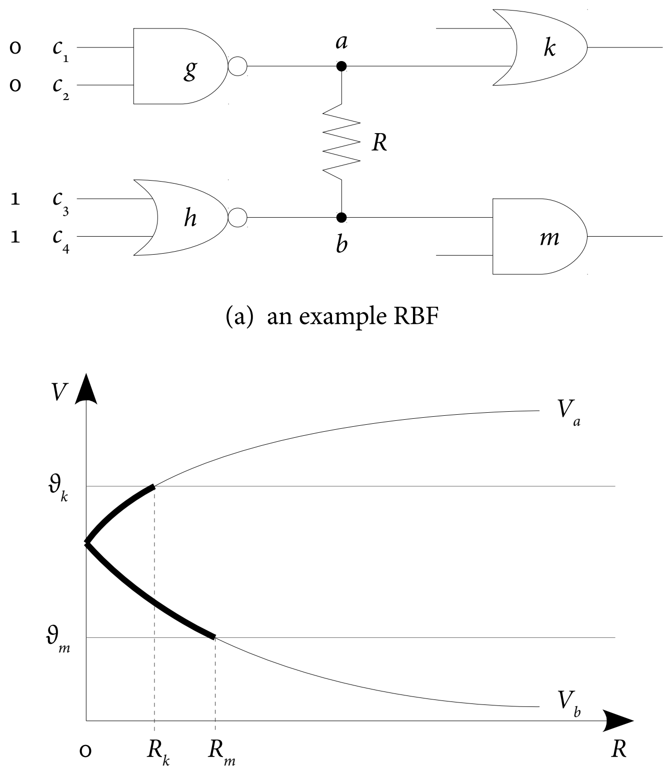
While these generic approaches have been proved to work well in modelling complex fault behaviour, there are cases in which not only the detection of faults is desirable, but also the satisfaction of additional optimisation goals. For example, maximising the number of primary outputs towards which the fault effect is propagated has been shown to increase the coverage of transition delay faults [247], while the minimisation of the number of fault-affected primary outputs is used to allow for better diagnosis [124].

In order to enable the specification of such optimisation goals, the CMS@FM model was extended. The new fault model, called *enhanced conditional multiple-stuck-at* fault model (ECMS@), is defined in Section 7.3. Then, the chapter introduces the implementation details on ECMS@-based SAT-ATPG. Finally, the chapter reviews selected applications of the ECMS@FM. For each application, the adequate mapping from the original problem to the ECMS@FM is discussed in detail, and experimental data are evaluated.

7.2 CMS@-based SAT-ATPG for resistive-bridging faults

Bridging faults with non-zero bridge resistance may impact the behaviour of a digital circuit in a non-trivial way. As explained in Section 2.5, in general, a short defect with resistance R between an interconnect node a driven by a gate g and a node b driven by a gate h induces voltages V_a and V_b, $V_a, V_b \in [0, V_{\mathrm{DD}}]$, on the affected nodes. These voltages depend not only on the actual resistance R, but also on the electrical parameters of the transistors within the gates g and h and on the number of transistors that are activated. Hence, V_a and V_b depend on the pattern that is applied to the inputs of g and h. Figure 25 shows an example, where the pattern 00 is applied to the inputs c_1 and c_2 of gate g and the pattern 11 is applied to the inputs c_3 and c_4 of gate h.

How the voltages V_a and V_b are interpreted by the gates k and m which are driven by a and b depends on threshold values between 0V and V_{DD}, which are defined individually for each gate and each input of the gate. In the example, the threshold ϑ_k determines whether gate k interprets the voltage V_a as logic 0 or logic 1. If $V_a < \vartheta_k$, which holds if $R < R_k$, gate k interprets V_a as logic 0; otherwise, it interprets the



(a) an example RBF

(b) corresponding resistance intervals

Figure 25. Mapping of resistive bridging faults to CMS@ faults

voltage as logic 1. Analogously, if $V_b > \vartheta_m$, which holds if $R < R_m$, gate m interprets V_b as logic 1; otherwise, it interprets the voltage as logic 0.

In general, the continuous space of resistance values R can be partitioned into n *sections* $[R_1, R_2], [R_2, R_3], \ldots, [R_n, \infty]$, where $R_1 := 0 < R_2 < \cdots < R_n < \infty$, such that the logical behaviour of the circuit is identical for all R-values within each section [199, 225, 74].

In order to demonstrate that the CMS@ fault model can be employed to perform defect-based ATPG, Tiguan was used to generate test patterns for CMS@ faults derived from resistive-bridging faults. In particular, RBFs are a good example of a realistic fault model that requires the modelling of multiple simultaneous stuck-at victim lines.

For each circuit, 10,000 pairs of interconnects were selected randomly. The section partitioning was extracted from the internal data of the RBF simulator SUPERB [74] assuming the same technology parameters as in [74]. The following example illustrates the mapping from sectioning information to CMS@ faults. Assume that the application of the pattern 0011 to the lines c_1, c_2, c_3 and c_4 leads to the voltage distributions shown in Figure 25 (b). Given the thresholds ϑ_k and ϑ_m, this leads to the sectioning $[0, R_k], [R_k, R_m], [R_m, \infty]$. In the first section, $[0, R_k]$, the voltage V_a is interpreted as logic 0 by gate k, whereas the correct logic value in absence of the bridge is logic 1. Hence, node a can be regarded as having s-a-0 behaviour under the presence of the bridge. Analogously, node b can be regarded as having s-a-1 behaviour under the influence of the bridge. In the diagram, the presence of stuck-at behaviour is marked by the bold sections of the curves for V_a and V_b. Altogether, if the bridge is present and its resistance R belongs to the interval $[0, R_k]$, nodes a and b simultaneously display s-a-0 and s-a-1 behaviour, respectively. However, this is only valid for the regarded input combination 0011 applied to $c_1, \dots, c_4$, because different input combinations result in different voltage curves. In summary, the circuit's faulty behaviour under the presence of the bridge can be expressed for the section $[0, R_k]$ by one CMS@ fault:

$$f_1 : \text{if } [c_1 = 0, c_2 = 0, c_3 = 1, c_4 = 1] \; a \text{ s-a-0}, b \text{ s-a-1}.$$

Analogously, the circuit's faulty behaviour for the section $[R_k, R_m]$ can be modelled by the CMS@ fault

$$f_2 : \text{if } [c_1 = 0, c_2 = 0, c_3 = 1, c_4 = 1] \; b \text{ s-a-1},$$

while the last section, $[R_m, \infty]$, is disregarded, as both nodes have voltages that are interpreted as fault-free behaviour for R-values greater than R_m.

The same analysis needs to be repeated for all input combinations that can be applied to $c_1, \dots, c_4$. In the end, the analysis produces a set of CMS@ faults $\{f_1, f_2, \dots, f_n\}$, where each of them corresponds to one section and one input pattern to be applied to the gates that drive the bridged nodes. Each of these CMS@ faults can be processed separately by TIGUAN without further knowledge of the technology parameters. A test set that detects all faults $f_1, f_2, \dots, f_n$ is guaranteed to detect the original RBF independently of the actual resistance that is present in an affected circuit. Note that some of these CMS@ faults may be undetectable. However, that does not render the original RBF undetectable. As explained in Section 2.5, the concept of detectability and fault coverage is more complex for resistive fault models. Instead of regarding an RBF as categorically detectable or undetectable, the RBF's detection probability is determined depending on how many sections can be detected, and depending on the probability that the actual resistance R lies in each section.

Table 26

CMS@-based SAT-ATPG with 32-bit fault dropping for RBFs — iscas'85, iscas'89 and itc'99 circuits

		classification			run-time (s)	
circuit	faults	detected	undetectable	patterns	avg/flt	total
c1355	20,433	5,169	15,264	610	0.0016	32
c1908	30,338	12,906	17,432	492	0.0013	40
c2670	28,276	17,661	10,615	2,045	0.0009	26
c3540	28,459	17,715	10,744	1,226	0.0019	54
c5315	28,214	19,594	8,620	1,661	0.0008	24
c6288	33,603	20,086	13,517	1,320	0.0037	125
c7552	32,028	19,024	13,004	1,224	0.0013	42
cs01196	19,647	10,510	9,137	819	0.0004	8
cs01238	19,532	9,997	9,535	824	0.0005	9
cs01423	23,767	16,681	7,086	862	0.0003	7
cs01488	13,589	8,964	4,625	336	0.0002	2
cs01494	13,480	8,586	4,894	321	0.0002	2
cs05378	28,929	27,388	1,541	1,520	0.0001	3
cs09234	21,835	15,853	5,982	1,514	0.0010	21
cs13207	20,366	15,107	5,259	1,115	0.0007	14
cs15850	20,061	14,803	5,258	1,090	0.0014	28
cs35932	27,160	9,332	17,828	133	0.0015	42
cs38417	25,976	20,174	5,802	1,619	0.0011	27
cs38584	26,602	17,207	9,395	1,486	0.0012	32
b13c	14,889	4,126	10,763	314	0.0002	3
b14c	44,837	7,692	37,145	2,236	0.0060	268
b15c	41,514	6,542	34,972	1,932	0.0119	495
b17c	41,651	7,966	33,685	2,925	0.0142	591
b18c	42,881	8,753	34,128	3,926	0.0250	1,070
b20c	44,378	8,073	36,305	2,285	0.0104	462
b21c	44,915	8,027	36,888	2,293	0.0104	468
b22c	44,824	8,551	36,273	2,170	0.0108	483

The experiment was carried out for iscas'85 circuits and for the combinational cores of iscas'89 and itc'99 circuits, as well as for the industrial nxp circuit suite, on a 2.3 GHz AMD Opteron computer with 4 Quad-Core processors (hence, up to sixteen computation threads can run in parallel, each on an own processor core) and 64 GB RAM, as usual with the limitation of 4 GB memory per Tiguan process independently of the number of SAT solving threads.

Table 26 summarises the results obtained for iscas and itc'99 circuits, and Table 27 shows the results obtained for nxp circuits. The test generation was combined with

Table 27

CMS@-based SAT-ATPG with 32-bit fault dropping for RBFs — nxp circuits

		classification				run-time (s)	
circuit	faults	detected	undetectable	aborted	patterns	avg/flt	total
p35k	21,809	19,985	1,824	–	6,325	0.0419	913
p45k	25,491	20,249	5,242	–	2,454	0.0026	67
p77k	23,580	17,738	5,841	1	3,413	0.1824	4,300
p78k	28,007	23,687	4,320	–	489	0.0016	46
p81k	31,144	22,703	8,441	–	3,618	0.0068	211
p89k	24,837	22,202	2,635	–	5,338	0.0050	123
p100k	26,077	21,381	4,696	–	2,687	0.0038	100
p141k	24,099	20,567	3,532	–	2,794	0.0235	567
p267k	22,504	19,645	2,859	–	3,519	0.0093	209
p269k	22,377	19,683	2,694	–	3,588	0.0093	207
p286k	25,676	20,478	5,198	–	3,901	0.0186	477
p295k	22,771	18,979	3,792	–	4,927	0.0143	327
p330k	23,716	20,991	2,725	–	4,428	0.0216	511
p378k	27,898	23,659	4,239	–	529	0.0060	166
p388k	24,637	21,495	3,142	–	2,139	0.0112	275
p469k	45,528	13,444	31,837	247	774	0.4523	20,594
p951k	21,967	20,106	1,861	–	1,958	0.0149	327
p1522k	22,731	19,167	3,564	–	5,731	0.0522	1,187
p2927k	22,638	19,351	3,286	1	3,761	0.0634	1,434

32-bit fault dropping (with random filling), and a timeout of 20 seconds per fault was imposed on SAT solving. No multi-threading was used. The second column of these two tables shows the number of targeted faults, which equals 10,000 multiplied by the average number of sections per RBF in which a fault effect needed to be encoded as CMS@ fault. The columns labelled *detected* and *undetectable* indicate how many faults were detected by pattern generation or by simulation, and how many faults were proved undetectable, respectively. Note that the undetectability numbers refer to the derived CMS@ faults and not to the original RBFs. The realistic detection probability of the original RBFs was not computed as this was out of the scope of this experiment.

The column labelled *aborted* quotes the number of faults that were left unclassified due to a SAT solving timeout. No faults were aborted for the circuits in Table 26; hence, a column *aborted* is not included in that table. As for the nxp circuits, all but three circuits were processed without aborts; two circuits have only one abort, and p469k has a number of aborts which stands out among all circuits, which however amounts to only 0.54% of all faults. Given the large number of undetectable faults found in this circuit, it is probable that these aborts correspond to further undetectable RBF sections.

The column *patterns* shows the amount of generated test patterns, which is smaller than the number of detected faults due to the use of fault dropping. For some circuits, like p469k, the number of patterns is significantly smaller than the number of detected CMS@ faults (i.e. RBF sections) despite the fact that no compaction techniques were employed. This shows that a large number of sections was detected by simulation.

The last two columns quote the average run-time needed to process each fault (this time comprises SAT formulation, SAT solving and fault simulation times), and the total run-time needed to process the whole fault list. Like in the experiment with CMS@ faults for gate-exhaustive testing (see also Tables 10 and 11), the average time needed to process each fault is higher than the average run-time measured for stuck-at faults (see also Tables 4 and 5), but it roughly remains in the same order of magnitude. The average run-time per fault amounts to only 0.003759 seconds for the ISCAS and ITC'99 circuits, and to 0.049511 for the NXP circuits (0.016105 without considering the hard benchmarks p77k and p469k), which allowed TIGUAN to process a very large number of CMS@ faults in reasonable total time.

The two-stage approach introduced in Section 5.1.2 was also applied to this set of CMS@ faults. The first stage was applied to all NXP circuits using an aggressive SAT solving time limit of 1 second per fault. In the second stage, TIGUAN was applied to the faults that were aborted in the first stage, however using a higher timeout of 20 seconds per fault and employing thread parallelism. Table 28 (a) summarises the results for circuits with at least one abort during the first stage. The second and third columns quote the run-time of the first stage and the number of faults aborted during the first stage, respectively. This is also the number of faults targeted in the second stage. The remaining columns are organised in three groups reporting the performance of the second stage using multi-threaded SAT solving with 1, 2 and 4 threads. The columns labelled *abr* quote the number of faults that remained unclassified after the second stage, while the columns labelled *time* and *total* quote the total run-time in seconds of the second stage and the accumulated run-time of both stages, respectively.

Table 28 (b) compares the best results observed for the two-stage approach to the results obtained by the one-stage approach with a timeout of 20 seconds and without a timeout (there are no aborts in this case). The experiment without timeout was only performed for circuits where not all faults could be classified with a timeout of 20 seconds. The second column indicates the number of threads employed during the second stage that leads to the results quoted in the third and fourth columns.

Table 28
Two-stage thread-parallel CMS@-based SAT-ATPG for resistive-bridging faults

(a) two-stage approach

	first stage		second stage (20s timeout)								
	(1s timeout)		1 thread			2 threads			4 threads		
circuit	time (s)	abr	abr	time (s)	total (s)	abr	time (s)	total (s)	abr	time (s)	total (s)
p77k	2,379	948	5	3,468	5,847	–	1,337	3,716	–	945	3,324
p141k	848	1	–	1	849	–	1	849	–	2	850
p330k	850	2	–	6	856	–	2	852	–	2	852
p469k	13,430	1,956	216	11,047	24,477	134	5,453	18,883	113	4,797	18,227
p1522k	1,695	6	–	12	1,707	–	6	1,701	–	6	1,701
p2927k	2,220	23	1	76	2,296	1	65	2,285	1	62	2,282

(b) comparison to one-stage approach

	best results of			one-stage approach		
	two-stage approach			20s timeout		no timeout
circuit	threads	aborted	time (s)	aborted	time (s)	time (s)
p77k	4	–	3,324	1	4,300	4,671
p141k	2	–	849	–	567	761
p330k	4	–	852	–	511	772
p469k	4	113	18,227	247	20,594	34,847
p1522k	4	–	1,701	–	1,187	2,090
p2927k	4	1	2,282	1	1,434	2,358

As was the case for stuck-at faults and for CMS@ faults for gate-exhaustive testing, the two-stage method is advantageous for several circuits. In nearly all cases, a higher number of employed threads translates into better run-times and less faults left unclassified. However, with exception of the hard benchmarks p77k and p469k, the difference in performance between the second stage with two SAT solving threads and the second stage with four SAT solving threads is minimal, which corroborates the general observations made in Section 5.1.2. Yet the two-stage approach offers the best compromise between number of aborts and run-time.

Also an experiment without SAT solving timeout was executed successfully. Tiguan was able to classify all RBF sections in all industrial circuits within acceptable times, which again shows the applicability of the Tiguan framework to complex fault models.

7.3 The ECMS@ fault model

In this section, the *enhanced conditional multiple-stuck-at* fault model (ECMS@) is defined. This model is an extension of the CMS@FM and replaced it as the internal fault model used in Tiguan.

The description of an ECMS@ fault is composed of three elements: the set of *condition lines* $A := \{a_1, \ldots, a_r\}$, which may be empty, the set of *victim lines* $V := \{v_1, \ldots, v_s\}$, and an optional *optimisation constraint*.

A *condition* c_{a_i} is imposed on each condition line a_i, and each victim line v_j has a fixed stuck-at behaviour s-a-b_{v_j}. Tiguan supports two classes of conditions that can be combined in arbitrary ways. The first one is the class of *hard conditions* which encompasses the following:

- 0 — a test pattern p satisfies this condition on a line l if the application of p to the primary inputs of the circuit induces the logic value 0 on l;
- 1 — p satisfies this condition on l if p induces the logic value 1 on l;
- F — p satisfies this condition on l if l is *fault-affected* under p, i.e. if the application of p propagates a fault effect through l;
- NF — p satisfies this condition on l if l is *non-fault-affected* under p, i.e. if the application of p propagates no fault effect through l.

An ECMS@ fault is activated under any input vector that satisfies all hard conditions; then, each victim line v_j behaves as s-a-b_{v_j}. A test pattern that detects such a fault needs to satisfy all conditions imposed on condition lines, to excite at least one victim line v_j (i.e. set its fault-free value to $\neg b_{v_j}$), and to propagate the error produced on at least one of the excited victim lines to a primary output.

The ECMS@FM also allows for the specification of additional optimisation goals which are expressed in function of the second class of conditions that can be imposed on condition lines, the class of *soft conditions*:

- $P0$ — the ATPG process has to try to generate a test pattern that induces the logic value 0 on lines with this condition;
- $P1$ — the ATPG process has to try to induce the logic value 1 on lines with this condition;
- PF — the ATPG process has to try to propagate a fault effect through lines with this condition;
- PNF — the ATPG process has to avoid propagating a fault effect through lines with this condition.

Since all these conditions have a P (which stands for *preferred*) in their names, these conditions will also be referred to as *P-type conditions*. A P-type condition on a line l instructs the ATPG process to *try* to satisfy the corresponding non-P-type condition on l. The intuition behind trying is that the ATPG process shall pursue the detection of the fault as primary goal, but if there is more than one test pattern that detects the fault as defined by its hard conditions and the behaviour of its victim lines, then the ATPG process shall give preference to test patterns that also satisfy the fault's soft conditions. For example, assume that the sectioning analysis of an RBF determines that one of two sections needs to be tested, and assume that the first section is valid if the pattern 1111 is applied to the inputs $c_1, \ldots, c_4$ of the gates whose outputs are bridged, while the second section is valid if the pattern 0111 is applied to those lines. Assume that the victim lines behave equally in both sections, but the first section is preferred over the second because it covers a wider range of resistances. Then, an ECMS@ fault that models this situation would declare c_2, c_3 and c_4 as condition lines with hard 1-conditions and c_1 as a condition line with a soft P1-condition. Then, the ATPG process will try to find a solution that assigns the logic value 1 to c_1, and search for solutions that assign the value 0 to c_0 only if no such solution exists.

As a means to increase the flexibility of this model, also an additional optimisation constraint can be imposed on the number ω of P-type conditions that are satisfied by a test pattern. For instance, ω can be maximised or minimised, or ω can be forced to lie between some application-specific bounds, or ω can be forced to be even or odd. With all these options, the ECMS@FM provides a flexibility that cannot be achieved using previously introduced fault models. Moreover, the framework can be easily extended by integrating new types of more specific basic conditions and new types of optimisation constraints.

For better legibility, an ECMS@ fault with hard conditions $\{c_{l_1}, \ldots, c_{l_r}\}$ imposed on lines $\{l_1, \ldots, l_r\}$, soft conditions $\{c_{l_{r+1}}, \ldots, c_{l_{r+t}}\}$ imposed on lines $\{l_{r+1}, \ldots, l_{r+t}\}$, and victim lines $\{v_1, \ldots, v_s\}$ where each victim line v_j has the behaviour stuck-at-b_{v_j}, is written as follows:

$$\text{if } [l_1\ c_{l_1}, \ldots, l_r\ c_{l_r}]\ v_1 \text{ s-a-} b_{v_1}, \ldots, v_s \text{ s-a-} b_{v_s}\ [l_{r+1}\ c_{l_{r+1}}, \ldots, l_{r+t}\ c_{l_{r+t}} : \text{constraint on } \omega].$$

For example, if $[l_1 = 1]\ v$ s-a-0 $[l_2\ PF, l_3\ PF, l_4\ PF : \text{maximise } \omega]$ stands for a fault that is activated if line l_1 is set to logic 1, in which case line v displays s-a-0 behaviour. Additionally, the ATPG process shall try to propagate it through as many of the lines l_2, l_3 and l_4 as possible.

The new type of conditions made available with the introduction of this fault model can be applied in a large number of different fields. One example is the test generation for *fault tolerant*[31] circuits that have been designed employing *information redundancy*. Such circuits operate on data encoded using *error-detecting* or *error-correcting* codes [141]. Such codes contain a redundant amount of data, based on which it is possible to detect or to correct up to a certain number of errors in the base data. In addition to the normal functional logic, self-checking circuits contain also *checking logic* that checks whether the inputs and outputs of the circuit are valid code words.

ATPG for such circuits needs to generate test patterns that are valid code words. Moreover, depending on the used coding scheme, the checking logic may also be able to automatically correct up to a certain number of errors contained in the circuit's output. Hence, the generated test patterns must produce valid code words also in the faulty case. Otherwise, some fault effects would be masked by the checking logic, and the corresponding faults would escape detection.

Special applications such as arithmetic circuits can be efficiently encoded using complex codes, like Berger or Bose-Lin codes [156, 104], but general applications are usually encoded with low-cost linear or parity codes [193]. For parity codes, for instance, the key to generating responses that are valid code words lies in manipulating the exact number of primary outputs towards which the fault effect is propagated, or in specifying the parity of sub-sets of primary outputs. These constraints are easily expressed by declaring the primary outputs as lines with *PF* or *PNF*-conditions and imposing the respective constraint on the number ω of satisfied *P*-type conditions.

Further applications of the ECMS@FM are discussed and evaluated in Section 7.5.

7.4 Implementation of ECMS@-based SAT-ATPG

The main Tiguan framework presented in Section 4.4 was extended in order to support ECMS@-based ATPG. The SAT solver antom was employed as SAT solving back-end, as antom's capability to solve SAT problems with qualitative preferences (see Section 3.3.4) is the mechanism that was utilised to enforce the satisfaction of soft conditions.

In a large part, the main CMS@-based framework remained unchanged, i.e. the core algorithms that colour the modelled circuit part, that assign Boolean variables to circuit lines and that generate SAT clauses are the same. Hard 0 and 1-conditions

[31] See Section 9.1 for a more detailed overview of various fault tolerance measures.

are equivalent to CMS@ aggressors that need to be set to 0 or 1, respectively. Hence, enforcing that a test pattern satisfies a 0 or a 1-condition is achieved by adding clauses that require that the G-variable of each line with one of these conditions is assigned to the appropriate value by any Boolean assignment that satisfies the SAT formula. Analogously, enforcing the satisfaction of F and NF-conditions is done by imposing the appropriate values on the D-variables that have been assigned to lines with these conditions.

In contrast, the enforcement of soft conditions is not performed by adding new clauses to the SAT formula. Lines with soft conditions are treated like lines with hard conditions by the colouring algorithm such as to ensure that the parts of the circuit which have an influence on the imposed conditions become part of the model, but the conditions are passed to the SAT solver in form of qualitative preferences. That means, Tiguan does not need to modify the SAT formula. Instead, it passes the SAT solving function a set of literals that should be preferably assigned to 1, and the SAT solver is in charge of internally taking into account Tiguan's preferences. Thus, for every G or D-variable X that would need to be assigned to 1 and every G or D-variable Y that would need to be assigned to 0 in order to enforce a non-P-type condition, Tiguan declares X and $\neg Y$ as preferred literals in order to enforce the corresponding P-type conditions.

Finally, the ECMS@FM allows for the specification of an optional optimisation constraint imposed on the number ω of satisfied P-type conditions. The formalism of SAT solving with qualitative preferences also allows the user to define a partial ordering that determines which preferred literals are more important than others. This is the mechanism that Tiguan uses in order to enforce the satisfaction of constraints imposed on ω.

7.4.1 Maximisation and minimisation of ω

Let f be a fault, let φ_f be the SAT formula that Tiguan has generated for f, and let $\mathfrak{X}_f$ be the set of Boolean variables that occur in φ_f. Let $\mathfrak{Y} := \{Y_1, \dots, Y_n\} \subseteq \mathfrak{X}_f$ be the *target set* of variables. The aim of the following algorithm is to extend φ_f into a SAT formula φ'_f, and to generate qualitative preferences $(L, \prec)$, such that the model of φ'_f generated by the SAT solver under consideration of $(L, \prec)$ satisfies φ_f and assigns the largest possible number of variables in $\mathfrak{Y}$ to 1.

First, n sets of new Boolean variables $\mathfrak{N}_i := \{N^i_1, \dots, N^i_i\}$, $i = 1, \dots, n$, are introduced such that $\mathfrak{N}_i \cap \mathfrak{X}_f = \varnothing$ for all $i = 1, \dots, n$.

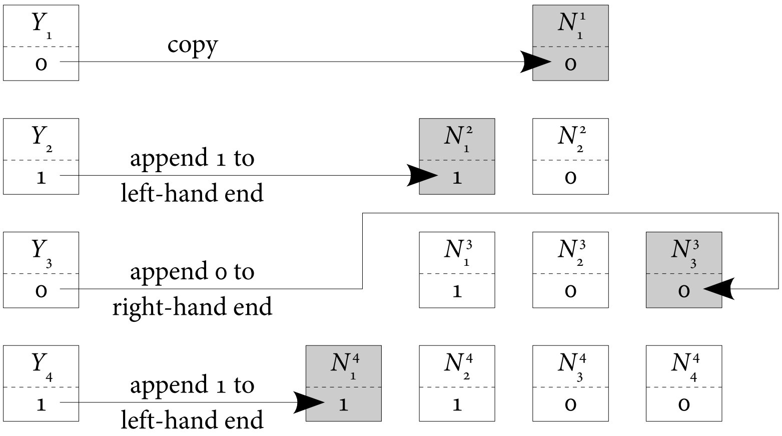


Figure 26. Sorting bit arrays using SAT

Then, the original SAT formula φ_f is extended such that any Boolean assignment w that satisfies the extended formula φ_f' meets the following conditions:

- $w(Y_1) = w(N_1^1)$;
- and for each $i = 2, \ldots, n$:
  - $(w(N_1^i), \ldots, w(N_i^i)) = (w(N_1^{i-1}), \ldots, w(N_{i-1}^{i-1}), 0)$ if $w(Y_i) = 0$, or
  - $(w(N_1^i), \ldots, w(N_i^i)) = (1, w(N_1^{i-1}), \ldots, w(N_{i-1}^{i-1}))$ if $w(Y_i) = 1$.

The intuition behind this method (see example for $n = 4$ in Figure 26) is that if w satisfies these conditions, then w assigns the N_j^i-variables values such that the sequences $N_1^1, N_1^2 N_2^2, \ldots, N_1^n \cdots N_n^n$ represent a step-by-step sorting of the 0 and 1-assignments made to the variables in $\mathfrak{Y}$. The first encountered value (Y_1) is copied into a sequence of length 1 (N_1^1). Then, every encountered 0-value is appended to the right-hand end of the sequence, while 1-values are appended to the left-hand end of the sequence. In the end, the sequence $N_1^n \cdots N_n^n$ is composed of a block of 1s followed by a block of 0s, and the number of 1s and 0s is the same as in the input sequence $Y_1 \cdots Y_n$.

This sorting algorithm is added to the SAT instance φ_f in the form of new clauses. For each N_j^i-variable, clauses that describe its value in function of the appropriate Y and N_j^i-variables are generated. For instance, the value of N_1^1 has to be equal to the value of Y_1 in the above example. This is enforced by adding the clauses $\{\neg Y_1, N_1^1\}$ and $\{Y_1, \neg N_1^1\}$ to φ_f. Analogously, the value assigned to N_3^4 has to be equal to the value assigned to N_2^3 if Y_4 is assigned to 1, or it must be equal to the value assigned to N_3^3 if Y_4 is assigned to 0. This is enforced by the clauses $\{\neg Y_4, \neg N_3^4, N_2^3\}$, $\{\neg Y_4, N_3^4, \neg N_2^3\}$, $\{Y_4, \neg N_3^4, N_3^3\}$ and $\{Y_4, N_3^4, \neg N_3^3\}$.

In order to maximise the number of 1-values assigned to the variables in $\mathfrak{Y}$, it is necessary to maximise the number of 1-values assigned to the variables in $\mathfrak{N}_n$, which is achieved by passing the SAT solver the preferences $(\mathfrak{N}_n, N_n^n < \ldots < N_1^n)$. That means, all literals in $\mathfrak{N}_n$ should preferably be assigned to logic 1. However, since it will in general not be possible to assign all preferred literals to 1, the literals on the right-hand side are to be given precedence $(N_n^n < \ldots < N_1^n)$. Since the sequence $N_1^n \cdots N_n^n$ is constructed such that all variables to the left of any variable set to 1 are also set to 1, the number of assigned 1-values becomes larger if the last 1 in the sequence is further on the right.

The same basic algorithm is used for the minimisation of 1-assignments to the target variables. Minimising the number of assigned 1-values is equivalent to maximising the number of assigned 0-values. This is achieved with preferences $(\{\neg N_1^n, \ldots, \neg N_n^n\}, \neg N_1^n < \ldots < \neg N_n^n)$.

Going back to the ECMS@FM, for each P-type condition line a of a fault f, a new Boolean variable Y_a is introduced, and clauses are generated for Y_a such that Y_a is assigned to 1 if and only if the line a has its preferred behaviour. Then, maximising or minimising the number ω of satisfied P-type conditions in f is achieved by applying the algorithm introduced in this section to the target set $\{Y_a : a \text{ is a } P\text{-type condition line of} f\}$.

7.4.2 Forcing ω to lie between application-specific bounds

Given a set of target variables $\mathfrak{Y} := \{Y_1, \ldots, Y_n\} \subseteq \mathfrak{X}_f$, the second type of constraints that can be imposed on the number ω of variables in $\mathfrak{Y}$ assigned to 1, is whether ω lies between a pair of application-specific bounds, i.e. whether $m_l \leq \omega \leq m_h$ for some fixed $m_l, m_h \in \{1, \ldots, n\}$.

The sorting algorithm from Section 7.4.1 serves also as basis for the encoding of this type of constraint. For any $m \in \{1, \ldots, n\}$, the sequence $N_1^n \cdots N_n^n$ contains at least m 1-assignments if $N_1^n = \cdots = N_m^n = 1$, independently of the values assigned to $N_{m+1}^n, \ldots, N_n^n$. In addition, in order for the sequence to contain exactly m 1-assignments, also $N_{m+1}^n = \cdots = N_n^n = 0$ must hold.

Hence, testing whether $\omega \geq m_l$ is equivalent to testing whether $N_{m_l}^n = 1$. Thus, it suffices to either pass $N_{m_l}^n$ as only preference to the SAT solver (soft constraint on ω), or to add the clause $\{N_{m_l}^n\}$ to the SAT-instance (hard constraint on ω).

The condition $\omega = m_h$ is met if $N^n_{m_h} \wedge \neg N^n_{m_h+1}$ is satisfied. Hence, $\omega \leq m_h$ is equivalent to testing whether $\neg N^n_{m_h} \vee \neg N^n_{m_h+1}$ is satisfiable. Enforcing that this condition is satisfied is achieved by introducing a new variable that is equivalent to this expression and passing it as preference or fixed clause to the SAT solver.

7.4.3 Controlling the parity of ω

Given a set of target variables $\mathfrak{Y} := \{Y_1, \dots, Y_n\} \subseteq \mathfrak{X}_f$, also the parity of the number of variables in $\mathfrak{Y}$ assigned to 1 can be controlled.

This is easily achieved by defining a new variable N and applying Tseitin transformation to an imaginary n-input XOR gate with inputs $Y_1, \dots, Y_n$ and output N. Thus, if an even number of variables in $\mathfrak{Y}$ is assigned to 1, N is assigned to 0, while an odd number of Y_i-variables assigned to 1 implies N being assigned to 1. Hence, it suffices to declare either N or $\neg N$ as the only preferred literal (soft constraint on ω), or to add the clause $\{N\}$ or $\{\neg N\}$ to the SAT formula (hard constraint on ω).

7.5 Advanced applications of the ECMS@FM

This section presents advanced applications of the ECMS@FM and discusses the obtained experimental results. All measurements were performed on a 2.3 GHz AMD Opteron 64-bit computer with 64 GB RAM. In all experiments, an unlimited SAT solving time budget was assigned to every processed fault. Hence, all targeted faults were classified, but the total run-times are also influenced by the time needed for very hard instances.

7.5.1 Controlling the amount of fault-affected POs

The first advanced application reviewed in this section is the generation of test patterns that constrain the number of activated propagation paths. Maximising the number of fault-affected primary outputs behaves similarly to n-detection approaches [186, 38] and increases coverage of transition delay faults [247]. Also the minimisation of the number of fault-affected primary outputs finds application, for instance in diagnosis. For example, in [124], a better localisation of faults is achieved using test patterns that propagate fault effects towards single outputs.

In order to test ECMS@-based SAT-ATPG as an application to this problem, patterns for stuck-at faults were generated for larger ISCAS'85 and for the combinational cores

Table 29

ECMS@-based SAT-ATPG for stuck-at faults — controlling the amount of fault-affected primary outputs

		fault-affected POs (%)			run-time (s)		
circuit	faults	basic	min	max	basic	min	max
c2670	594	38.4	35.8	91.7	0.8	1.1	1.5
c3540	601	32.9	18.2	81.7	2.2	5.7	5.5
c5315	929	16.7	10.9	97.7	2.3	3.5	3.9
c6288	1,488	7.8	6.2	99.7	17.3	21.2	1,142.5
c7552	1,408	27.8	15.9	98.8	7.6	11.8	9.3
b14c	2,401	8.1	2.9	40.8	33.0	100.8	4,911.2
b15c	2,989	4.5	1.3	35.2	86.4	512.8	149,851.4
b17c	9,155	7.0	1.6	41.7	516.8	1,583.7	416,221.3
b18c	28,395	3.6	1.7	32.6	2,844.9	3,821.1	817,311.4
b20c	5,090	4.8	2.2	33.1	167.5	249.9	102,311.0
b21c	5,187	5.6	2.2	32.2	215.3	302.1	107,265.7
b22c	7,397	6.4	2.2	34.1	311.4	503.8	1,858,574.3
avg/sum		13.6	8.4	59.9	4,205.5	7,117.5	3,457,609.0

of ITC'99 circuits under the additional constraint of maximising or minimising the number of fault-affected primary outputs. Expecting the resulting SAT instances to be extremely hard due to the large amount of preferred literals, only faults at the roots of fan-out-free regions were targeted.

For each targeted stuck-at fault v s-a-b, the ECMS@ fault

$$v \text{ s-a-} b \left[o_1^v\ PF, \dots, o_n^v\ PF : \text{maximise } \omega \right]$$

was processed by Tiguan, where $o_1^v, \dots, o_n^v$ are the primary outputs in the output cone of v. This means that all primary outputs that can be affected by an error on v are declared via P-type conditions as lines that should be preferably fault-affected and the maximisation of the number of satisfied P-type conditions is requested. For the minimisation of the number of fault-affected primary outputs, only the optimisation constraint [maximise ω] needs to be replaced by [minimise ω].

Table 29 lists the obtained results. The first and second columns quote the circuit names and the number of targeted faults, respectively. The column group labelled *fault-affected POs* show the percentage of primary outputs that displayed a fault effect under the generated test patterns. The group *run-time* lists the total run-time needed to process all faults. In both column groups, the columns labelled *basic* show the results of normal ATPG without maximisation or minimisation constraints, while the columns labelled *min* and *max* list the numbers achieved when the number of fault-affected outputs was minimised or maximised, respectively. The last table row labelled *avg/sum* shows the average percentage of fault-affected primary outputs in all circuits and the accumulated run-time needed to process all circuits.

The optimisation ATPG is able to minimise the average percentage of fault-affected POs from 13.6 to 8.4%, while the maximisation experiment achieves very high percentages (up to 99.7%). Note that these numbers are guaranteed to be the best achievable percentages, because ANTOM was extended such as to return the optimal model under the given set of preferences according to the optimality definition introduced in Section 3.3.4. However, the satisfaction of the minimisation and maximisation goals results in a run-time increase which essentially stems from the increased SAT-solving time due to the handling of qualitative preferences. For instance, the average SAT solving time per fault measured for circuit b22c increased from 0.0051 seconds to 0.0325 seconds when minimisation was applied, and to 251.1935 seconds when maximisation was applied.

The run-time increment is especially severe in the maximisation experiments, where the difference in the number of fault-affected POs is larger. Although it must be remarked that the algorithm that the current ANTOM version uses to handle preferences was not tuned for efficiency, the disproportionate growth of the SAT solving time is not only attributable to the large number of literals declared as preferred. Maximising the number of fault-affected outputs means maximising the number of paths along which the fault effect is simultaneously propagated. This triggers a high amount of reasoning along paths that would otherwise be ignored by an ATPG algorithm that is only advised to find any propagation path as fast as possible. And this applies not only to SAT-based, but also to structural algorithms. Hence, this is an inherently hard optimisation problem.

Aiming for better run-times, the maximisation experiment was repeated, but in this case only half of the primary outputs were declared as preferred. The percentage of fault-affected primary outputs and the needed run-time are quoted in the columns labelled *max/2* in Table 30. The columns labelled *max* quote the same numbers as in Table 29 and are included here for comparison. As can be seen, the achieved percentages are nearly as high as in the first experiment. These numbers are optimal given the set of preferences, but they are not optimal given the original problem any more, as only half the primary outputs were explicitly constrained. However, the achieved percentages represent a good compromise between optimality and run-time efficiency, given that the required time falls to less than 10% of the time needed during the first experiment. This shows that the way in which original problems are mapped to ECMS@-ATPG has a very strong influence on the optimality of

Table 30

ECMS@-based SAT-ATPG for stuck-at faults — run-time-efficient, nearly-optimal control of the amount of fault-affected primary outputs

	fault-affected POs (%)		run-time (s)	
circuit	max	max/2	max	max/2
c2670	91.7	88.8	1.5	1.2
c3540	81.7	73.1	5.5	3.4
c5315	97.7	86.8	3.9	3.0
c6288	99.7	97.1	1,142.5	1,108.0
c7552	98.8	94.7	9.3	7.9
b14c	40.8	35.1	4,911.2	573.7
b15c	35.2	30.3	149,851.4	12,484.9
b17c	41.7	35.3	416,221.3	51,350.3
b18c	32.6	28.0	817,311.4	75,449.1
b20c	33.1	29.0	102,311.0	6,651.3
b21c	32.2	28.4	107,265.7	5,861.2
b22c	34.1	30.0	1,858,574.3	61,715.9
avg/sum	59.9	54.7	3,457,609.0	215,209.9

the achieved results and on the required run-time. This also demonstrates the strength of ECMS@-based ATPG, which provides the flexibility to solve related but not identical ATPG problems without the need to modify the overall SAT-ATPG framework.

7.5.2 SAT-ATPG with control of switching activity

The second ECMS@ application discussed in this chapter is the generation of tests for some basic fault model while inducing specific switching behaviour on a number of observed lines in the vicinity of the fault location. For instance, slow-down-crosstalk testing [30] requires that a number of aggressor lines switch in the *opposite* direction in which the victim line switches, such as to increase the fault effect. In contrast, the creation of maximal noise stemming from the chip's power distribution network necessitates the induction of transitions in the *same* direction on the outputs of the gates connected to the same segment of the network. As a consequence, the power supply voltage of the involved gates is reduced and their propagation delay increases due to *power starvation* [182]. In this context, minimising the switching activity on other lines in the circuit intensifies the desired effect.

This ECMS@ application was tested by generating tests for transition faults while controlling the switching activity on neighbouring lines of the fault site. The *transition fault* model (TFM) is a simple static fault model that assumes that there is a faulty gate that cannot produce a rising or a falling transition. Transition faults can also be regarded as GDFs with an infinite delay [129]. Similarly to other delay fault models (see Section 2.4.3), the test of a TF requires the application of two test patterns — an initialisation pattern that brings the CUT into a known and stable state, and a propagation pattern that makes the fault effect visible. For example, a test pair that detects a rising-transition fault on the output l of a gate g has to justify the value 0 on l in the first cycle and the value 1 in the second cycle in order to excite the fault, and the second pattern has to propagate the fault effect to a primary output.

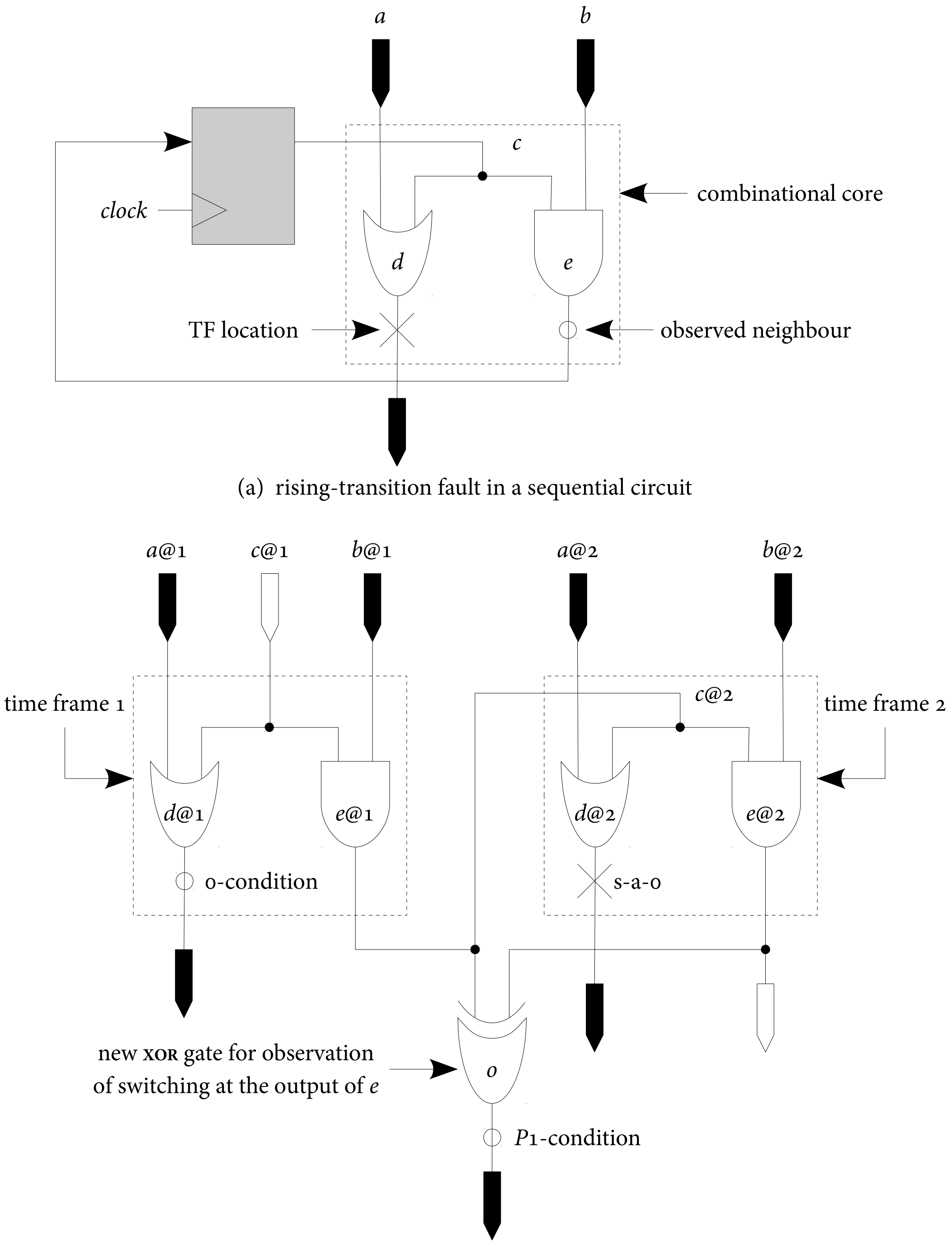
Given a combinational or sequential circuit C and a rising-transition fault on a line v, m neighbour lines $a_1, \dots, a_m$ were chosen. Since no layout and technology data were available for this experiment, the neighbours were chosen randomly among all lines on the same topological level as v. Hence, under the unit-delay assumption, the fault location and all neighbours switch at the same time.

The mapping to an ECMS@ fault was done as follows. Assuming a full-scan and launch-off-capture testing mode, a two-time-frame expansion C' of the original sequential circuit is constructed. Let l be a line in C. Then, $l@1$ and $l@2$ denote the lines in C' which correspond to l in the first and in the second time frame, respectively. An example is shown in Figure 27.

In order to minimise the switching activity on the neighbours, for each neighbour a_i, $i \in \{1, \dots, m\}$, a new *observation gate* of type XOR with inputs $a_i@1$ and $a_i@2$ and output o_i is added to the sequential expansion. This means, whenever a neighbour switches, the output of its corresponding observation gate is 1. Hence, the minimisation of switching on all observed neighbours can be achieved by enforcing the assignment of the value 1 to a minimum number of observation gate outputs. The resulting ECMS@ fault in C' is the following:

$$\text{if } [v@1 = 0] \ v@2 \text{ s-a-0 } [o_1 \ P1, \dots, o_m \ P1 : \text{minimise } \omega].$$

The hard 0-condition on v in the first time frame and the s-a-0 ECMS@ victim on v in the second time frame constitute the conditions to the detect the transition fault independently of any optimisation goals, and the soft conditions on the observation gates enforce that the found pattern induces the minimum amount of switching on the neighbours. No conditions are imposed on the off-path inputs in the first time frame. While none are explicitly imposed in the second time frame either, a propagation of the stuck-at effect in the second time frame is only possible if all



(a) rising-transition fault in a sequential circuit

(b) sequential expansion

Figure 27. SAT-ATPG for transition faults with control of switching activity on neighbours — mapping to ECMS@ faults

Table 31

ECMS@-based SAT-ATPG for transition faults — minimising the switching activity of neighbours

		switch (%)		run-time (s)	
circuit	faults	basic	min	basic	min
s01196	528	25.1	1.0	4.6	7.4
s01238	508	24.0	0.5	4.3	7.1
s01423	650	34.2	7.8	11.4	20.1
s01488	652	30.7	12.1	13.3	31.6
s01494	645	30.9	12.6	13.2	31.8
s05378	2,779	28.9	0.4	177.9	288.2
s09234	5,597	22.1	1.8	798.9	702.0
s13207	7,948	17.7	0.9	1,150.3	764.5
s15850	9,772	14.0	1.3	2,155.5	2,035.1
s35932	16,065	32.2	2.1	1,405.6	1,868.8
s38417	22,173	14.0	0.1	6,396.2	8,705.3
s38584	19,245	19.2	3.0	6,366.3	4,863.7
b12	874	8.3	1.7	25.9	25.0
b13	244	15.1	5.8	0.9	1.1
b14	5,346	20.5	0.3	1,770.2	1,560.7
b15	7,022	15.2	0.6	6,086.8	6,355.2
b17	22,757	12.7	0.2	39,465.2	36,861.9
b18	69,913	13.8	0.0	257,210.3	244,630.2
b20	11,948	16.0	0.1	12,520.0	10,987.4
b21	12,125	15.0	0.1	11,111.7	10,791.0
b22	17,326	15.5	0.1	23,498.9	20,461.8
avg/sum		20.2	2.5	370,187.4	350,999.9

off-path inputs of the propagation path are set to the non-controlling value of their respective gate. Hence, these conditions will be satisfied by the SAT solver. Overall, this corresponds to strong non-robust (see Section 2.4.3) test generation for the original transition fault.

The experiment was carried out targeting the rising-transition fault at every gate output of sequential ISCAS’89 and ITC’99 benchmark circuits. For every fault location, the switching activity on 10 neighbours was minimised. The results are listed in Table 31. The column group *switch* quote the percentage of observed neighbours that underwent a falling or rising transition under the application of the generated test patterns. The single-column labels have the same meaning as in Table 29. Again, the achieved percentages of satisfied *P*-type conditions are optimal. In contrast to the experiment from Section 7.5.1, however, the total run-time is on average lower

Table 32

ECMS@-based SAT-ATPG for transition faults — controlling the transition direction of neighbours

		same-direction transitions				opposite-direction transitions			
		switch (%)		run-time (s)		switch (%)		run-time (s)	
circuit	faults	basic	max	basic	max	basic	max	basic	max
s01196	528	12.1	56.5	4.0	4.4	12.8	57.1	4.1	5.6
s01238	508	11.7	54.5	3.9	4.1	12.1	56.7	4.0	4.7
s01423	650	19.2	65.6	10.5	9.9	16.2	66.3	11.2	12.8
s01488	652	15.3	34.3	12.5	17.3	15.3	37.9	11.0	15.7
s01494	645	15.3	35.1	11.8	19.1	15.4	37.5	10.4	15.8
s05378	2,779	13.6	84.8	156.9	177.7	14.9	86.0	203.4	155.5
s09234	5,597	11.9	82.2	693.9	717.7	10.2	81.7	691.1	685.4
s13207	7,948	9.3	81.4	828.0	879.6	8.4	81.5	866.5	844.0
s15850	9,772	7.7	80.2	2,172.7	2,503.0	6.1	80.5	2,308.9	1,970.4
s35932	16,065	22.0	91.8	1,882.6	1,663.2	10.3	90.9	1,537.5	1,592.0
s38417	22,173	6.9	95.3	7,693.4	6,117.3	7.2	95.4	6,983.4	7,924.6
s38584	19,245	10.7	83.4	4,422.5	5,245.3	8.5	82.4	4,301.9	4,692.9
b12	874	4.2	24.8	31.7	35.3	4.3	24.9	31.4	40.3
b13	244	7.9	57.0	1.1	1.4	6.8	50.4	1.2	1.5
b14	5,346	11.0	71.8	1,932.5	2,348.6	9.3	68.6	1,460.8	2,679.6
b15	7,022	7.9	45.7	5,537.3	7,068.7	6.9	43.0	5,865.0	8,657.6
b17	22,757	6.3	65.6	40,152.7	41,375.1	6.2	64.3	40,250.4	41,419.1
b18	69,913	7.0	81.0	248,677.0	220,822.2	6.8	79.5	254,168.1	236,724.5
b20	11,948	8.1	80.5	12,222.1	12,487.0	7.9	79.7	11,772.7	13,088.3
b21	12,125	7.7	80.5	10,490.5	13,062.3	7.1	79.9	10,659.9	11,650.0
b22	17,326	8.0	83.7	19,857.4	22,102.2	7.6	83.7	23,815.4	24,541.6
avg/sum		10.6	68.3	356,795.0	336,661.4	9.5	67.9	364,958.3	356,721.9

when the optimisation goal is enforced. The reason for this is that the number of preferred literals is considerably smaller, and it also shows that a moderate number of preferences is often able to constrain the SAT problem in a way that speeds up the SAT solving.

Two further experiments were performed using these settings. In the first one, the number of aggressors switching in the same direction in which the victim switches was maximised, and the number of aggressors switching in opposite direction was maximised in the second. The construction of the circuit expansion is the same, but the observation gate of type XOR is replaced by gates of appropriate types, such that the output of each observation gate is 1 whenever an observed internal line behaves as desired. The results of these two experiments are summarised in Table 32. The column groups *switch* quote the percentage of observed neighbours that underwent a transition in the desired direction under the application of the generated test patterns. The single-column labels have the same meaning as in Table 31. Like in the experiment presented in Section 7.5.1, it could be expected that the enforcement of the maximisation constraint would increase the run-time due to the large difference in the number of satisfied P-type conditions. However, the average run-time is on average lower, which corroborates the observation that SAT-ATPG with preferences is not universally harder than SAT-ATPG without preferences.

7.6 Conclusions

The first part of this chapter discussed a further application of the CMS@FM which permitted the generation of test patterns for bridging faults with non-zero resistance, which are a very good example of a realistic defect model that requires the handling of multiple stuck-at victims. The test generation for RBFs was done by performing a sectioning analysis of the targeted bridges and by mapping the single sections to generic CMS@ faults, which allowed to take advantage of the benefits of SAT-based ATPG with regard to the processing of hard-to-detect faults. The experimental results demonstrated the applicability of the approach, as all targeted RBF sections were successfully classified, even in large industrial circuits. Also the two-stage approach previously introduced to speed up test generation for stuck-at faults was successfully applied to RBF-ATPG.

The second part of this chapter introduced an extension of the SAT-ATPG framework around the CMS@FM. The new model (ECMS@FM) allows to generically emulate highly complex defect modelling approaches; for instance, sophisticated aggressor-victim models that describe non-trivial signal integrity phenomena like crosstalk or power supply noise. The most important feature of the ECMS@FM is that it provides an easy-to-use method to specify optimisation constraints that guide the SAT-ATPG process towards the generation of preferred patterns. For example, test patterns were generated which sensitise the maximum achievable number of primary outputs, as well as test patterns that provide precise control of local switching activity. All these applications exemplify the type of non-trivial optimisation problems that are very difficult to solve using tools based on structural ATPG. Furthermore, the ECMS@-based approach provides a framework that is universally applicable, while previous non-generic approaches required the implementation and maintenance of individual tools for each considered problem.

While the test patterns that control local switching activity could be generated without incurring a loss of run-time efficiency, the generation of patterns that maximise the number of fault-affected primary outputs lead to a very important observation. Complex optimisation problems can be mapped to the ECMS@FM in different ways. While the generated solutions are always guaranteed to be optimal with respect to the chosen model at ECMS@ level, the mapping from the original problem to ECMS@-ATPG can be done with different levels of accuracy, and the used level of accuracy needs to be chosen carefully such as to achieve the best compromise between solution optimality and run-time efficiency. The next chapter presents the application of the ECMS@-ATPG framework to ATPG for power droop test, an extremely hard optimisation problem in which the global switching activity has to be controlled over a very large number of clock cycles, which serves as example to demonstrate the high relevance of appropriate ECMS@ mapping.

8

Power droop testing

Power droop is a non-trivial effect on signal integrity triggered by specific power supply conditions over a large number of clock cycles. Hence, ATPG for power droop testing is an extremely hard problem that has not yet been solved optimally using structural methods. This chapter discusses the use of ECMS@-based SAT-ATPG to generate test patterns for power droop test, which shall illustrate the versatility of the ECMS@ framework. After reviewing the physics behind power droop, the chapter briefly introduces a structural test generation method previously developed by the author of this thesis for his undergraduate studies (Studienarbeit). Then, the ECMS@-based test generation method is introduced and evaluated experimentally. In particular, important directions for future research are analysed in order to achieve further performance improvement for ECMS@-based SAT-ATPG.

Author's contribution — The author's contribution consisted in the application of the SAT-ATPG tool Tiguan to the extremely complex test generation problem for power droop testing, where the main challenge is posed by the question how to choose the optimisation criteria that best reflect the original ATPG problem without rendering the resulting SAT-ATPG instances impractically hard. The author explored several strategies and evaluated them experimentally.

Parts of the work covered in this chapter have been published in [J3, C18, C5, W2] (see author's publications on pages 223–226).

8.1 Introduction

The test of signal integrity is often associated with effects between logic lines. For example, *capacitive crosstalk* [149, 261, 143, 42], *inductive oscillatory noise* [228] and combinations of these effects [229] have been considered in the past. The generation of patterns to test a circuit's vulnerability to such effects needs to take the induction of specific signal behaviour, e.g. noise, into consideration. For example, noise is maximised on the gates along a sensitised path in [142]. In contrast, test sequences that avoid high noise levels are generated in [22, 251, 252]. However, these works consider only the noise induced by the transition of one single signal, while it has been shown that effects that stretch over several clock cycles cannot be neglected [123, 138].

Furthermore, signal integrity is not solely challenged by noise effects between logic lines. Also the characteristics of the power distribution network can lead to signal integrity errors. In [123], for instance, genetic algorithms are used to find test sequences that most accurately estimate the peak power over various time durations. Aside from peak power, also sharp changes in energy consumption are of concern. Such changes can occur during normal operation. For instance, a microprocessor that has been in idle mode for several cycles may considerably increase its power consumption within only few cycles when it starts complex calculations that involve simultaneous use of several functional units.

Ground bounce and power droop are two example effects of power consumption transients. *Ground bounce* [236, 39, 40, 41] denotes a phenomenon caused by voltage transients on the chip's power and ground, which result from rapid changes in current demand. Such transients can cause the ground potential on-chip to rise above the power supply ground potential. The rise may be sufficient to provoke *false highs*, i.e. voltage levels are erroneously interpreted as logic 1. If the rise is strong enough, it may even make V_{DD} appear to be negative, which leads to unstable states in circuit components, e.g. in flip-flops.

Power droop (PD) [245, 177] describes an even more complex mechanism that can result in transition delays. In [182], an ATPG method aimed at creating worst-case power droop conditions was proposed. The method tries to generate a long test sequence that induces two types of power droop effects in succession. The first effect, called *low-frequency* power droop, occurs when the voltage regulation module is unable to handle large transients in the power consumption of the whole device due to non-negligible inductance in the interconnect. It results in a drop of voltage on certain lines (*power starvation*), and it requires several clock cycles to cause the worst impact. In contrast, the source of the second effect, called *high-frequency* power droop, is chip-internal. It results from the limited capability of the power distribution network on-chip to deliver power quickly enough. If a number of lines supplied by the same power distribution node increase their power demand simultaneously, the power grid might not be able to deliver the required amount of current in appropriate time, which leads to delay effects.

The proposed test procedure applies three sub-sequences of test patterns in succession. The first sub-sequence induces minimum switching activity (and thus low power consumption) in the circuit under test, while the second sub-sequence maximises global switching activity. The induced transient then leads to low-frequency power droop. When its effects have reached their largest impact, a final test pair is applied which induces worst-case high-frequency power droop by imposing simultaneous signal transitions in the same direction on a victim line and on several aggressor lines that draw power from the same segment of the power grid. If the circuit is vulnerable to power droop, this leads to a delay effect on the victim line.

Since this test sequence needs to be applied in functional mode, using scan for any test vector of the sequence except for the first one is not possible. In consequence, the generation of such a test sequence constitutes sequential ATPG involving a large number of time frames. In addition, each sub-sequence has to minimise or maximise specific switching behaviour, either globally or on the set of aggressor lines. All these aspects make the ATPG problem for power droop test (PD-ATPG) an extremely hard ATPG problem.

In [182], the *dynamically constrained D-Algorithm* was proposed. The method works in a greedy fashion and gives lines in later time-frames preference over lines in earlier time frames when making search decisions. Hence, the overall solution is not optimal. Decisions made in order to augment the effect of high-frequency PD during later time frames may result in less severe low-frequency PD effects during earlier time frames, although a better global solution might exist with a weaker high-frequency-PD effect in later time frames and considerably better low-frequency-PD effects during earlier time frames.

In contrast, the ECMS@-based method presented in this chapter considers all modelled time frames simultaneously, which enhances the quality of the overall solution. It makes use of the SAT solver ANTOM's capability to solve SAT instances under a given set of qualitative preferences (see Section 3.3.4), thus guaranteeing the generation of optimal solutions with respect to the specified optimisation constraints. However, as shown in the previous chapter, the way in which the original problem is mapped to ECMS@-ATPG has a strong influence on TIGUAN's run-time performance. This means that a well-balanced set of criteria has to be chosen for the mapping in order to obtain solutions of sufficient quality without rendering the generated SAT instances impractically hard.

8.2 Power droop testing

This section briefly reviews the physics behind low-frequency and high-frequency power droop, as well as the test method that was proposed in [182].

Let L denote the parasitic inductance of the interconnect between the circuit under test and power supply. The sudden increase in current I demanded per time unit t, i.e. the sudden increase in power consumption, is referred to as a *dI/dt event*. After a dI/dt event, the circuit under test sees its power supply voltage V_{DD} reduced by $L \cdot dI/dt$, where dI is the amount by which the current demand increases and dt is the time within which that increase occurs. This effect is known as *low-frequency* power droop. In modern designs, where V_{DD} has been scaled down at the same time that the operation speed lies in the range of several GHz, the *inductive voltage drop* $L \cdot dI/dt$ is non-negligible. For example, on a chip operated at 3 GHz, a transient of 10W (dI = 10A for V_{DD} = 1V) occurring within 3 clock cycles (10^{-9} seconds) leads to a voltage drop of $L \cdot 10^{10}\frac{\mathrm{A}}{\mathrm{s}}$, which equals to 0.1V (10% of V_{DD}) for an inductance as low as 0.01nH. SPICE simulations have shown that a 10–15% voltage drop can cause gate delays to increase by 20–30% [132].

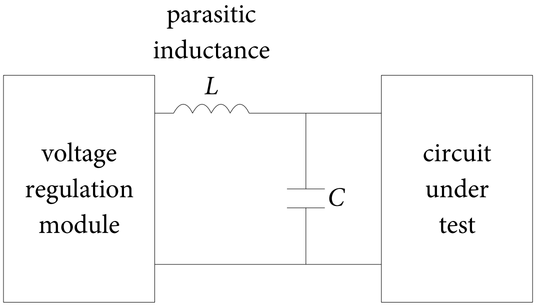


Figure 28. Circuit under test connected to power supply

The effect of low-frequency power droop can be diminished by adding a capacitance C intended to cover the circuit's short-term demand for current after a dI/dt event (see Figure 28). Since the inductance of the line between the capacitor and the circuit is lower than L, the inductive voltage drop is less severe. However, if C is discharged before the voltage regulation module is ready to supply the full amount of needed current, a V_{DD} drop is still observed, albeit of a smaller extent and some

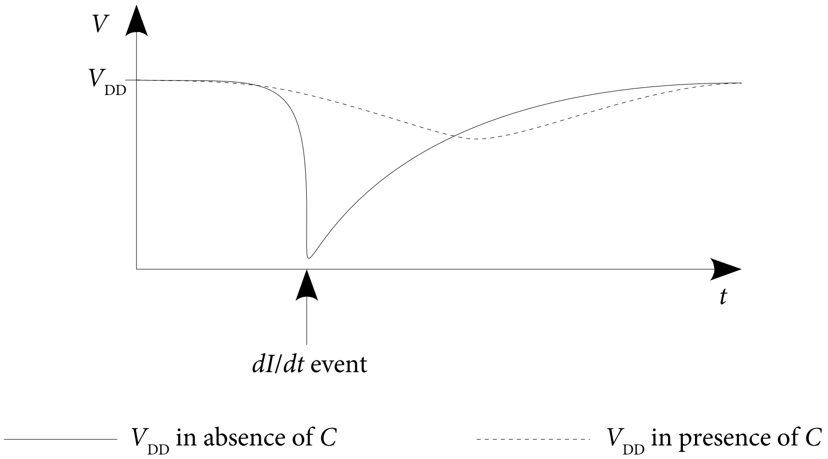


FIGURE 29. VOLTAGE SEEN BY CUT AFTER A *dI/dt* EVENT

time after the initial *dI/dt* event (see Figure 29). V_{DD} decreases more slowly, and after some time a point is reached at which the power supply provides enough current and starts recharging the capacitor. Whether the V_{DD} drop is large enough to cause a logic or delay failure depends on the extent of the *dI/dt* event, i.e. on how much more current is demanded over how small a period of time, and on the values of *L* and *C*, as well as on the characteristics of the voltage regulation module. However, the key observation is that the impact of low-frequency power droop is most severe several clock cycles after the actual *dI/dt* event.

As opposed to low-frequency power droop, *high-frequency* power droop is caused only by conditions inside the IC. The power distribution network or *power grid* of CMOS ICs is organised in multiple metal layers, where each layer is composed of several parallel *rails*, with vertical *vias* connecting rails of different layers (see Figure 30). The power rail part located between two vias is called a *segment*. The upper layers are often reserved for power rails and the clock-distribution network while lower layers are shared with logic signal lines. For this reason, rails and vias tend to be smaller in lower layers. In particular, vias are relatively small with respect to the rail size. In consequence, they are an obvious bottleneck for power delivery. High-frequency power droop occurs when multiple cells drawing current from the same power grid segment suddenly increase their current demand simultaneously. If the current cannot be provided quickly enough from other parts of the chip, a voltage drop is observed in the affected area. In contrast to low-frequency power droop, this is a highly localised and transient phenomenon: one of the involved cells is slowed down for one clock cycle.

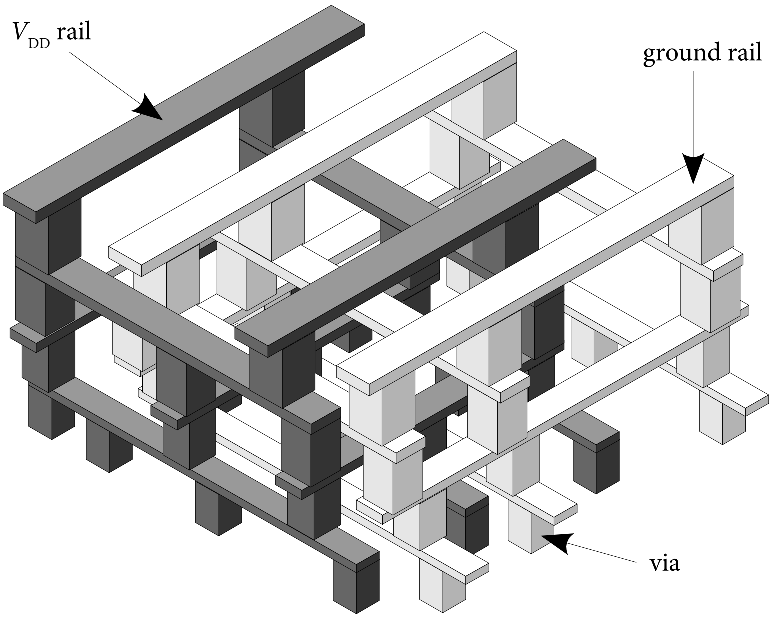


Figure 30. Four-layer power grid

The test method introduced in [182] is designed such as to maximise the effects of both low-frequency and high-frequency power droop. First, low-frequency power droop is induced by creating a global *dI/dt* event that stretches over multiple clock cycles. When the voltage droop is most severe due to the effects of low-frequency PD, high-frequency PD is imposed on a *victim line v*. For this purpose, line *v* and a number of *aggressor lines* that draw power from the same segment as *v*, are required to switch simultaneously and in the same direction. Combined power starvation due to both low-frequency and high-frequency power droop leads to an increased switching delay on line *v*. Finally, a path from line *v* to an output or flip-flop is sensitised, such that the fault effect can be observed.

Given a victim line *v* and a set of aggressor lines, the aim is to generate a test sequence $p_1, \dots, p_{M+1}, \dots, p_{M+N+1}, p_{M+N+2}$ that satisfies the following conditions:

- *condition* $\kappa 1$ — the last two test patterns p_{M+N+1} and p_{M+N+2} have to be a strong non-robust test for a transition fault located at *v*;
- *condition* $\kappa 2$ — the application of p_{M+N+1} followed by p_{M+N+2} has to induce a transition in the same direction in which *v* switches on as many aggressor lines as possible;

- *condition* $\kappa 3$ — the global switching activity has to be as low as possible during the application of the test sequence $p_1, \dots, p_{M+1}$; this sequence is referred to as *LSS* (low-switching-activity sequence);
- *condition* $\kappa 4$ — the global switching activity has to be as high as possible during the application of the test sequence $p_{M+1}, \dots, p_{M+N+1}$ (*HSS* — high-switching-activity sequence).

$\kappa 1$ is a necessary condition for the test, while $\kappa 2$, $\kappa 3$ and $\kappa 4$ are all optimisation conditions. The extent to which $\kappa 2$, $\kappa 3$ and $\kappa 4$ are satisfied will determine the quality of the generated test sequence.

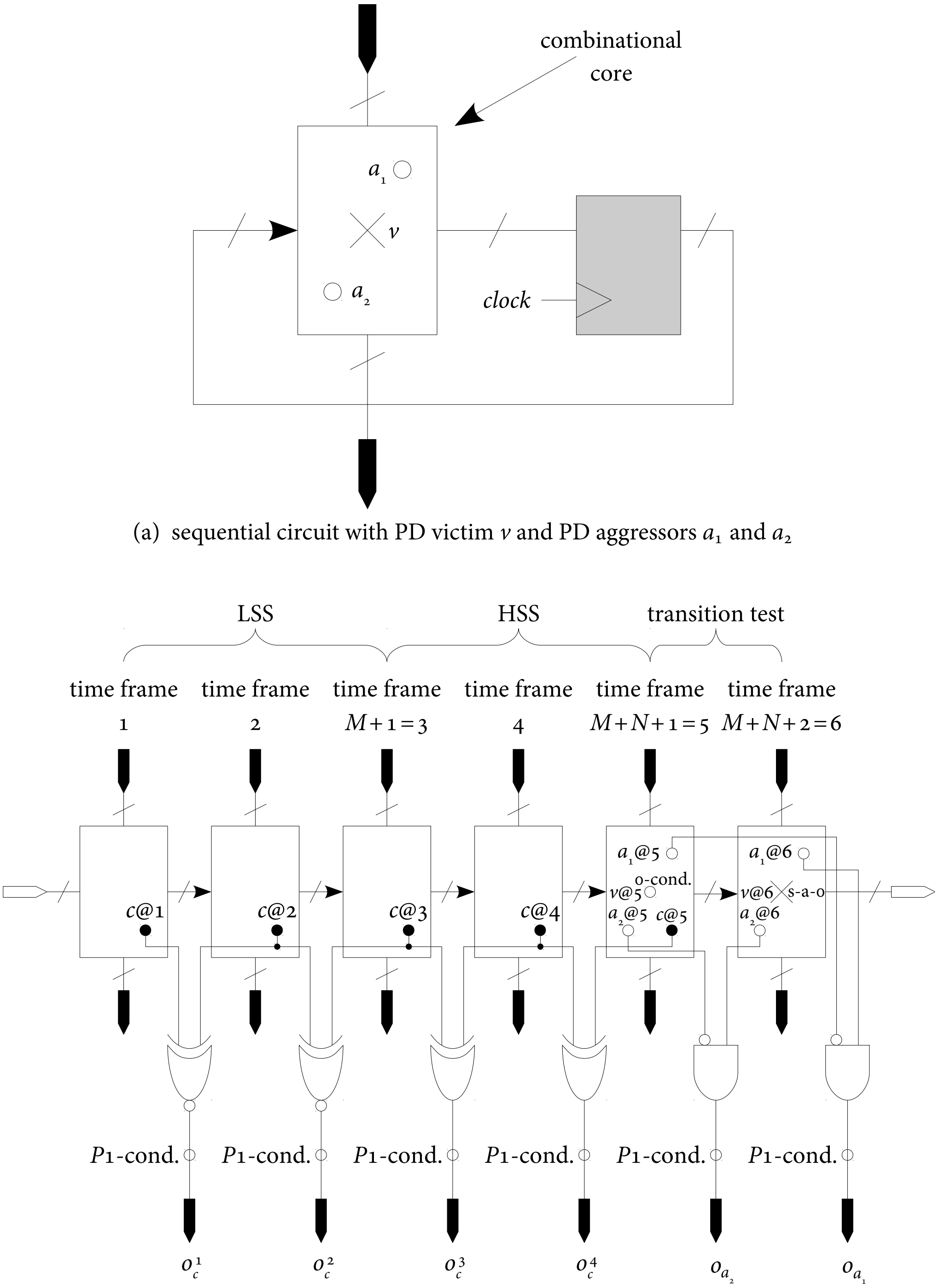
8.3 Mapping to ECMS@-based SAT-ATPG

Given the gate-level net list of a sequential circuit C, a victim line v and a set of aggressor lines, the aim is to generate a sequence of $M + N + 2$ test patterns that complies with the conditions $\kappa 1$, $\kappa 2$, $\kappa 3$ and $\kappa 4$ that were defined in the previous section. In order to simplify the description of the algorithm, it is assumed that the victim undergoes a rising transition. Figure 31 illustrates the mapping of PD-ATPG to ECMS@-ATPG for $M = N = 2$, i.e. for a total of six modelled time frames.

First, assuming that all secondary inputs are controllable in the first time frame, and that all secondary outputs are observable in the last time frame, an $M + N + 2$-time-frame expansion C' of the original sequential circuit C is constructed. For any line l in C, and $i = 1, \dots, M + N + 2$, let $l@i$ denote the line in the sequential expansion C' which corresponds to l in the i-th time frame.

Then, for each PD aggressor line a, a new AND gate with inputs $\neg a@M + N + 1$ and $a@M + N + 2$ is added to C'. The output o_a of this new gate will produce the logic value 1 if and only if a undergoes a rising transition under the application of the last two test patterns of the generated test sequence.

The next step consists in selecting a set of *control lines* in the original circuit C. Then, for each control line c, and $i = 1, \dots, M$, a new XNOR gate with inputs $c@i$ and $c@i + 1$ is added to C'. The output o_c^i of the new gate will produce the logic value 1 if and only if line c does not change its value (i.e. c does not switch) following the application of the i-th and the $i + 1$-th patterns of the generated test sequence. For $i = M + 1, \dots, M + N$, gates of type XOR are used instead of XNOR, such that the output of those new gates produce the logic value 1 when the corresponding control line c switches following the application of the i-th and the $i + 1$-th patterns. In the example in Figure 31, there is only one control line c.



(a) sequential circuit with PD victim v and PD aggressors a_1 and a_2

(b) sequential expansion for $M = N = 2$, i.e. a total of 6 time frames

FIGURE 31. SAT-ATPG FOR POWER DROOP TEST — MAPPING TO ECMS@ FAULTS

All AND, XNOR and XOR gates that have been added to C' in these first two steps are referred to as *observation gates*. On the whole, all observation gates are connected such that they produce the logic value 1 whenever some line in C exhibits the behaviour that is desired for that line at a certain time point during the test application.

Finally, given aggressor lines $a_1, \dots, a_r$ and control lines $c_1, \dots, c_s$, the constructed ECMS@ fault is as follows:

$$\text{if } [v@M+N+1=0] \; v@M+N+2 \text{ s-a-0}$$
$$[o^1_{c_1}\ P1, \dots, o^{M+N}_{c_1}\ P1, \dots, o^1_{c_s}\ P1, \dots, o^{M+N}_{c_s}\ P1, o_{a_1}\ P1, \dots, o_{a_r}\ P1 : \text{maximise } \omega].$$

The hard 0-condition on v in the last-but-one time frame and the s-a-0 ECMS@ victim on v in the last time frame constitute the conditions to detect the transition fault independently of any optimisation goals. The soft conditions on the outputs of all observation gates, together with the constraint of maximising the number ω of satisfied soft conditions, enforce that the found test sequence simultaneously:

- maximises the number of control lines that do not switch during the application of the sub-sequence LSS,
- maximises the number of control lines that switch during the application of the sub-sequence HSS,
- and maximises the number of PD aggressors that undergo a rising transition during the application of the last two test patterns.

Given that the SAT solver ANTOM guarantees to generate an optimal SAT model with respect to a given set of preferences according to the optimality definition presented in Section 3.3.4, the produced test sequence is also optimal with respect to the specific choice of control lines. However, the optimality of the test sequence with respect to the original problem depends on the strategy employed to choose the control lines.

The intuition behind a good strategy for the selection of control lines is that as few control lines shall be chosen, however such that controlling the switching activity of the chosen control lines has a sufficiently large influence on global switching activity. The reason why only few control lines may be selected is the large number of $P1$-conditions that need to be set, which equals $s \cdot (M + N) + r$. In a preliminary experiment, in which all circuit lines were selected as control lines, the large number of preferences made SAT solving infeasible even for small SAT formulae: average SAT solving times of more than 5,000 seconds per SAT instance were measured for a circuit with only 200 gates.

8.4 Experimental evaluation

This section discusses the experimental evaluation of the presented method. All measurements were performed on a 2.3 GHz AMD Opteron 64-bit computer with 64 GB RAM. In all experiments, an unlimited SAT solving time budget was assigned to every processed fault. Hence, all targeted faults were classified, but the total runtimes are also influenced by the time needed for very hard instances.

8.4.1 Comparison to structural ATPG

In a first experiment, the SAT-based approach was compared to the *dynamically constrained D-Algorithm* presented in [182]. As in [182], only one fault site per circuit was targeted, namely the node with the largest fan-out. Five aggressors were selected randomly among all lines in the vicinity of the victim. The selected aggressors belong all to the same level as the victim. Hence, the fault location and all aggressors switch at the same time under the unit-delay assumption. Like in [182], an LSS length M and an HSS length N of 10 were chosen, which yields a test sequence of length 22. For the SAT-based method, five randomly chosen FFR (fan-out-free region, see Section 2.2.2) roots were used as control lines.

The results are shown in Table 33. The columns labelled $\kappa 2$ quote the number of aggressors (max. 5) that complied with test condition $\kappa 2$, i.e. that switched in the same direction in which the victim switched. The columns labelled $\kappa 3$ list the percentage of all lines in the circuit that switched in any direction during the application of LSS; according to test condition $\kappa 3$, low percentages are desired in these columns. The columns labelled $\kappa 4$ quote the percentage of all lines in the circuit that switched in any direction during the application of HSS; in these columns, larger numbers are better. Finally, the columns labelled *time* report the needed processor time in seconds.

Since the SAT solver returns the optimal solution under the given set of preferences, TIGUAN's solution to the problem of satisfying all three conditions $\kappa 2$, $\kappa 3$ and $\kappa 4$ is optimal as well, whereas the structural approach from [182] is heuristic and not guaranteed to return the optimal solution. This is clearly reflected by the numbers reported in the table. For nearly all circuits, TIGUAN achieves better numbers for all three conditions. However, since the employed ECMS@ mapping gives the same weight to all conditions, there are cases in which single conditions are better satisfied by the structural algorithm, e.g. $\kappa 2$ for circuit s00444. Here, TIGUAN makes only one aggressor comply with $\kappa 2$. In contrast, the difference in switching activity between

Table 33

ECMS@-based SAT-ATPG for power droop test — comparison to structural ATPG

	structural [182]				Tiguan			
	satisfied conditions				satisfied conditions			
circuit	κ2	κ3 (%)	κ4 (%)	time (s)	κ2	κ3 (%)	κ4 (%)	time (s)
s00027	1	0	70	0	5	1	53	0.2
s00208	3	3	40	1	5	2	81	1.1
s00298	2	4	46	1	1	17	69	1.9
s00344	1	0	46	1	3	0	58	1.3
s00349	1	4	47	2	3	1	58	1.4
s00382	2	2	34	1	2	14	76	5.5
s00386	1	2	38	3	3	26	70	1.2
s00400	2	3	35	2	2	13	82	25.7
s00420	4	0	36	6	4	0	87	5.3
s00444	2	2	40	2	1	13	81	22.3
s00510	2	16	39	7	0	6	70	5.8
s00526	5	1	37	3	3	14	77	14.8
s00641	5	5	55	5	5	1	75	1.0
s00713	5	2	53	6	5	1	74	1.9
s00820	1	17	42	12	3	17	65	0.9
s00838	2	0	32	4	2	4	95	6.8
s00953	4	1	34	103	4	2	77	8.0
s01238	1	0	39	19	1	9	70	0.8
s01423	2	1	40	35	5	16	75	9.7
s01488	2	8	34	96	2	6	63	10.6
s01494	2	14	31	246	3	6	63	12.7
s05378	2	4	55	9,024	5	23	78	150.1
s09234	4	8	44	4,465	5	18	72	355.6

the application of LSS and HSS achieved by Tiguan amounts to 68, while the D-Algorithm achieves a difference of only 38. Hence, the solution found by Tiguan is likely to induce more suitable conditions for power droop testing.

As opposed to ATPG for simple fault models like the SAFM, where structural approaches are faster than SAT-based algorithms except for very hard instances, the run-times reported in Table 33 show that SAT-based ATPG is more suitable for complex fault models. In this case, where not only the fault model is more complex, but where optimisation objectives are also to be satisfied, the SAT-based approach performs considerably better than the structural approach in terms of run-time. By expressing the optimisation constraints by means of qualitative preferences, the complexity of the optimisation problem is passed to the SAT solver. The sophisticated learning techniques allow the SAT solver to find an optimal solution more quickly than the dynamically constrained D-Algorithm.

8.4.2 Evaluation of strategies for the selection of control lines

In a second series of experiments, different strategies for the choice of control lines were evaluated. The aim is to find a strategy that produces the best quality of patterns while avoiding an explosion of SAT solving time. The method was applied to ISCAS'89 and ITC'99 benchmark circuits.

An interesting finding of [245] is that the number of possible sites for high-frequency power droop is very limited — less than 100 for a microprocessor of 128,000 standard gates. Based on this experience and on the expected hardness of the problem, only up to 100 power droop victims per circuit were considered. In general, the number of possible power droop victims depends on the actual layout of the circuit. Determining realistic victim and aggressor candidates requires a thorough analysis of the power grid. A higher load on a power grid segment results in a higher probability that one of the lines connected to that segment is affected by high-frequency power droop. No such analysis was performed for the experiments reported here, as no layout data was available at the time of execution. The choice of the victims to be targeted was done based on the gate-level net list's topology. The benchmark circuits were partitioned into fan-out-free regions, and their root gates were targeted as PD victims, as these nodes are more likely to have the largest load. If there were more than 100 FFR roots, the 100 FFR roots with the largest fan-out were chosen.

In [138], time stretches of several dozen time frames are considered. However, given the problem's hardness and that the proposed algorithm generates optimal solutions, the number of time frames was limited to 22 time frames, like in the first experiment.

The first strategy that was tested consisted in choosing primary or secondary inputs as control lines. First, all primary or secondary inputs in the victim's input cone are determined. Then, a number of these inputs are defined as control lines, choosing those which influence the largest number of signals. Table 34 (a) compares the results that were generated using this strategy for different numbers of control lines. Column *strat* shows the fraction of primary or secondary inputs taken as control lines in the corresponding table row. For instance, *1/2* means that half the inputs found in the victim's input cone were chosen as control lines.

The second column shows the number of targeted power droop sites (victims). Column $\kappa 2$ lists the percentage of aggressors that complied with test condition $\kappa 2$, i.e. that switched in the same direction in which the victim switched. Column $\kappa 3$ lists the percentage of all lines in the circuit that switched in any direction during the application of LSS (lower percentages are better), and column $\kappa 4$ lists the percentage

Table 34
ECMS@-based SAT-ATPG for power droop test — choice of control lines

(a) using PIs/SIs as control lines

			satisfied conditions (%)			run-time (s)	
circuit	faults	strat	$\kappa 2$	$\kappa 3$	$\kappa 4$	slv/flt	total
s00344	28	1/16	30.8	10.1	23.8	0.05	2
		1/8	23.9	9.8	27.3	0.22	7
		1/4	33.7	8.9	27.6	1.25	37
		1/2	33.6	8.5	27.8	7.49	211
s00400	19	1/16	36.3	11.5	15.2	0.32	7
		1/8	38.5	11.5	16.1	0.60	12
		1/4	35.8	11.5	17.4	6.39	123
		1/2	34.8	11.1	17.6	152.31	2,895
s00444	23	1/16	40.1	12.6	17.1	0.27	7
		1/8	43.9	12.4	17.6	0.54	14
		1/4	44.8	10.4	18.2	7.80	181
		1/2	41.6	10.9	18.9	480.59	11,056
s00713	63	1/16	27.3	4.0	11.6	0.22	22
		1/8	29.9	7.9	19.8	0.59	47
		1/4	33.0	5.5	22.5	45.21	2,863
s00953	99	1/16	33.0	8.7	18.9	0.88	103
		1/8	34.2	9.6	20.9	21.87	2,181
		1/4	39.2	8.0	20.9	261.56	25,917
s05378	100	1/16	36.3	16.7	21.4	72.44	7,380
		1/8	35.8	16.8	21.5	537.01	53,839

(b) using internal lines as control lines

			satisfied conditions (%)			run-time (s)	
circuit	faults	strat	$\kappa 2$	$\kappa 3$	$\kappa 4$	slv/flt	total
s00713	63	1/8	29.9	7.9	19.8	0.59	47
		agg	38.5	6.6	17.9	0.78	59
		ffr	24.0	5.2	28.4	1.10	79
s00953	99	1/8	34.2	9.6	20.9	21.87	2,181
		agg	37.3	9.5	23.9	14.75	1,483
		ffr	31.6	5.1	25.1	24.01	2,392
s05378	100	1/8	35.8	16.8	21.5	537.01	53,839
		agg	36.5	16.4	21.5	39.36	4,073
		ffr	26.6	24.7	25.3	14.78	1,668

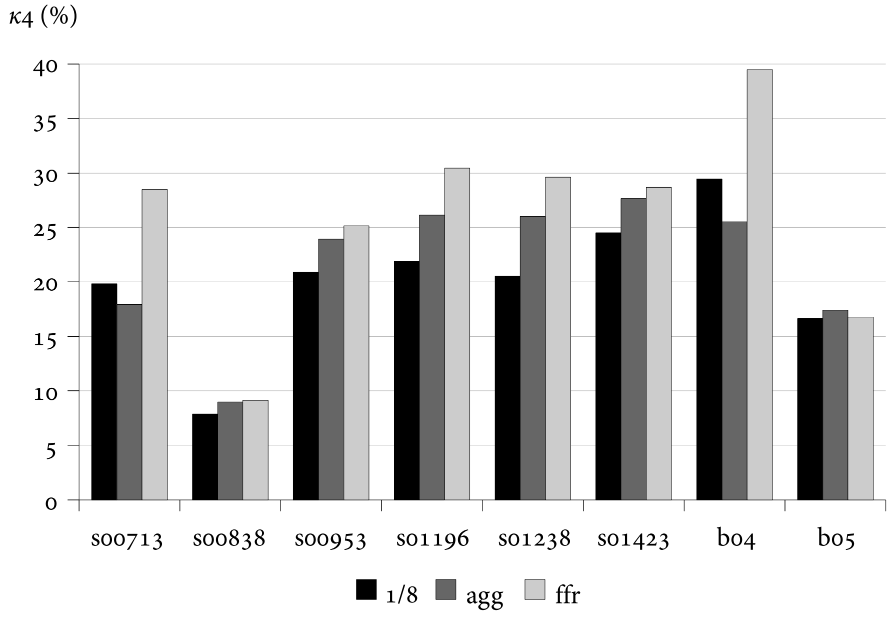
of all lines in the circuit that switched in any direction during the application of HSS (higher percentages are better). These numbers represent the average over all faults. The last two columns list the measured run-times in seconds. Column *slv/flt* lists the average SAT solving time per instance, while total run-times are quoted in column *total*.

Note that columns $\kappa 3$ and $\kappa 4$ list the percentage of lines in the whole circuit that exhibited switching activity, not the percentage of lines with P-type conditions that were satisfied. The percentage of satisfied P-type conditions is not listed in the table, as it is in the same order of magnitude (more than 90% in the average) for all strategies and it does not directly reflect the quality of the generated test patterns.

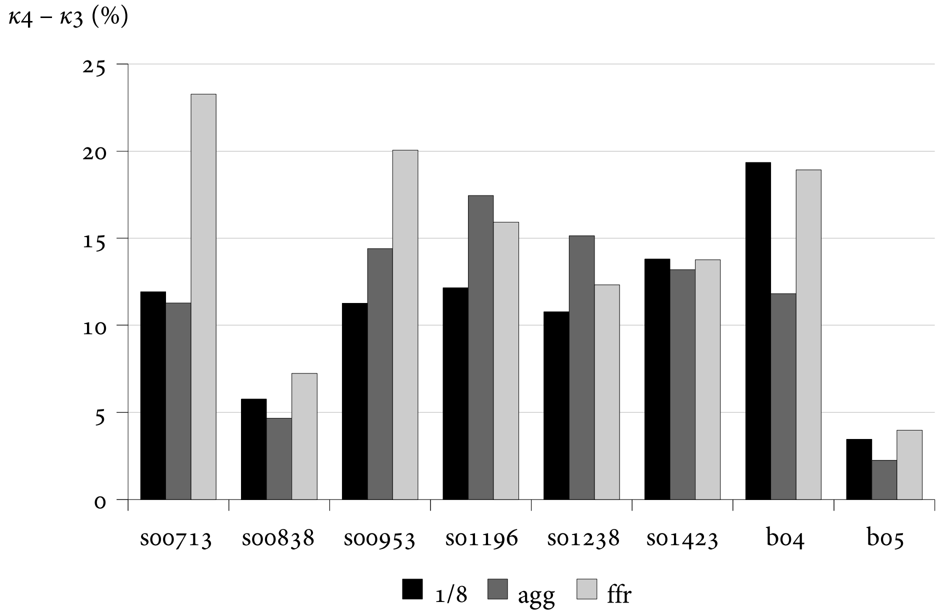
The quality of the generated test sequences is measured by the compliance with rules $\kappa 2$, $\kappa 3$ and $\kappa 4$, i.e. a strategy is better than another if the number in column $\kappa 2$ is larger, and if the number in column $\kappa 3$ is smaller while the number in column $\kappa 4$ is greater. As expected, using more control lines leads to better test quality, however at the expense of heavily increased SAT solving times. As explained at the end of Section 8.3, increasing the number of control lines not only increases the SAT instance's size, but imposes more qualitative preferences on the SAT solving, thus compelling the SAT solver to the generation of more Boolean solutions until finding the optimal one. In addition, using more control lines does not necessarily constitute the best choice. Consider e.g. circuit s00400. In average, the 1/2-mode needs 23.8 times more time for the solution of an average SAT instance than the 1/4-mode. However, the quality of the solution improves only minimally. Hence, a different kind of strategy is required.

The second class of strategies that were considered consist in using internal circuit lines as control lines. Results for example circuits are listed in Table 34 (b). The column labels have the same meaning as in Table 34 (a). Column *strat* shows which strategy was used in the corresponding table row. *1/8* is the same strategy from Table 34 (a), listed here for comparison. *agg* means that the PD aggressor lines were used as control lines. *ffr* means that 5 randomly chosen FFR roots were used as control lines. As can be seen, the strategies that use internal lines for the control of switching activity are better than the strategies that use circuit inputs. In particular, the FFR-based strategy results in visibly better $\kappa 4$-numbers. At the same time, the SAT solving times are considerably better. This shows that the number of control lines seems to have the largest influence on run-time, while an intelligent selection of control lines has an influence on pattern quality.

Figure 32 (a) illustrates the $\kappa 4$-numbers produced by the same three strategies for several other ISCAS'89 and ITC'99 circuits. The FFR-based strategy's results are better than those of the other two strategies, thus corroborating the previous observations.



(a) percentage of lines that switch during HSS application



(b) percentage of lines that switch during HSS application, but not during LSS application

Figure 32. ECMS@-based SAT-ATPG for power droop test — choice of control lines

In all cases, the run-time of the FFR-based strategy was also better than the run-time of the other two strategies.

In addition, the difference between the switching activity during LSS and HSS application ($\kappa 4 - \kappa 3$) was considered. A larger difference stands for a stronger low-frequency PD effect. As shown in Figure 32 (b), the strategies that use internal lines of the circuit for control of global switching activity achieve better pattern quality.

8.5 Conclusions

In order to demonstrate the versatility of the ECMS@-based test generation framework for problems with additional optimisation constraints, the framework was used to generate test patterns for power droop testing. ATPG for power droop constitutes an extremely hard variant of sequential test generation, given that three different optimisation objectives need to be satisfied simultaneously.

The experimental results show the applicability of the method to mid-sized benchmark circuits. In particular, the analysis performed in this chapter demonstrates the relevance of intelligent mapping from the original problem to ECMS@-ATPG. For any given ECMS@-based model of the original problem, the produced solution is guaranteed to be optimal. But different ways of mapping can lead to significant differences in run-time performance and test quality with respect to the original problem.

In order to permit the use of more sophisticated strategies and the application to large industrial circuits, the basic algorithm presented in this chapter can be improved with respect to different aspects. One important direction for future research is the incorporation of heuristic methods into the SAT solver, such that the minimisation or maximisation of the number of Boolean literals set to specific values can be achieved by carefully guiding the SAT solver's search instead of employing preferences. Solution optimality would be the expense of such a heuristic extension, but it has been shown both in Chapter 7 and in this chapter that nearly-optimal solutions can be obtained with significantly reduced run-times if the mapping is done based on carefully chosen circuit lines.

Regarding test generation for power droop ATPG, future research should include an electrical evaluation of the quality of the generated test sequences, e.g. by means of SPICE simulation.

9

Applications to process variations and fault tolerance

The aim of this chapter is to present a further area of research in which SAT-based test generation methods have found application. First, the chapter discusses the basic principles behind the interface of a C++ library that was implemented in order to allow other researchers to integrate Tiguan as a SAT-ATPG back-end engine into their client applications. Then, the text reviews two works carried out by fellow doctoral students at other universities, where the Tiguan library was successfully deployed.

Author's contribution — The author's contribution consisted in the development of a C++ library that allowed other researchers to use Tiguan as a SAT-ATPG back-end engine. Also, extensive documentation and support for their applications was provided.

Parts of the work covered in this chapter have been published in [C14, C8, W6, W4] (see author's publications on pages 223–226).

9.1 Introduction

As was explained in Chapter 1, modern, nano-scale CMOS technologies have become increasingly difficult to control, for example due to the use of new materials with different properties. With these innovations, also the topic of *process variations* has strongly gained in importance lately. This term refers to variations that occur among the population of manufactured circuits due to the natural variability of the manufacturing process. For example, any given gate usually has minimally different nominal delays in two different copies of the same circuit, because the exact delay depends on several parameters like the actual transistor dimensions. Process variations have a large impact on the performance of circuits, and this impact is expected to increase even further [28, 25, 8, 9].

Variability can be addressed using design techniques [106, 191, 44, 61, 58, 62], or self-adaption and fault tolerance [139, 174, 150, 154]. Fault tolerance refers to measures to design *robust* or *fault-tolerant* circuits, i.e. circuits that can tolerate (i.e. detect or correct) a certain amount of errors. The main principle of robust circuit design is redundancy, which can be implemented in the form of time, hardware or information redundancy [141].

Hardware redundancy, for instance, means that components that are more prone to fail are either replicated or designed using redundant structures such that the failure of some of these structures can be tolerated up to a certain extent. A well-known example is *triple modular redundancy* (TMR) [141], which consists in triplicating a component and adding a *voter* module that decides that component's correct output in function of the outputs produced by the majority of the three component instances. However, TMR implies a high hardware cost in terms of area and power consumption. Thus, TMR is often implemented at component rather than at system level. That means, only critical components are triplicated. A gate or a design block are said to be *critical* if excessive variation affecting them is likely to result in circuit misbehaviour. Further examples of the application of fault tolerance measures only to critical components is the use of *error-resilient* flip-flops [260] or *selective hardening* [175, 262].

Another possibility to counter the cost of hardware redundancy is the use of *information redundancy*, which means that the circuit is designed such as to operate on data encoded using *error-detecting* or *error-correcting* codes [141]. Such codes contain a redundant amount of data, based on which it is possible to detect or correct up to a certain number of errors in the base data. Circuits designed in this way are called *self-checking*. In addition to the normal functional logic, self-checking circuits contain also *checking logic* that checks whether the inputs and outputs of the circuit are valid code words. In order for the checking logic to serve its purpose, undesired component sharing between the functional and the checking logic has to be avoided during synthesis. However, this cannot always be controlled when synthesis is automated using CAD tools. Moreover, designers may also decide to give up some checking ability in order to reduce the overall hardware cost. Hence, there is a need to analyse the synthesised logic's secureness, i.e. the degree to which it will be able to detect and correct errors. Section 9.3 presents an approach to reduce this problem to ATPG. The work was performed by Marc Hunger at Paderborn University [126] and employed Tiguan as a SAT-ATPG back-end engine.

Test tools that address variability also need to adapt the concepts of detectability and fault coverage. In the case of gate delay variations, for example, the variation may be very small for single gates, but it occurs in all gates. Hence, the accumulated delay along a path can vary sufficiently such that a path that is the longest sensitisable path in a specific circuit instance is not the longest path in a different circuit instance [52]. For path-based delay testing, this means that a larger number of paths need to be identified because different circuit instances need to be considered (the same net list, but varying nominal delays). In consequence, the efficient search of longest sensitisable paths is a topic that is receiving considerable attention [190, 209]. In Section 9.4, a method that optimises the KLPG-Algorithm [190] is presented. The work was performed by Jie Jiang at Passau University [131] and also employed Tiguan as a back-end engine.

9.2 The Tiguan library

In order to give the users of the Tiguan engine maximum flexibility, the author of this thesis implemented a C++ library [2] with an advanced interface. This section reviews the most relevant details on the Tiguan library.

Aside from flexibility, the most important feature of the library interface is that it allows for a tight integration between the client application and the Tiguan library. For instance, all data are passed to the Tiguan library via dedicated functions and do not need to rely on run-time-expensive file system operations.

In order to ease the use of the library, the main concept behind the implemented interface is the encapsulation of all ATPG-related data into an object that is managed by the library. Thus, the client application communicates with the library solely over that central object and does not need to take care of tasks like the collection of statistics (total and average test generation run-times, SAT formula sizes, etc.) and the management of memory for the data structures that represent, for example, Tiguan's internal fault list, test set, original circuit and, if required, the sequential expansion of the original circuit.

For each aspect of the test generation process, the Tiguan library provides a set of functions that allow the client application to interact with the library independently of how Tiguan's internal data are managed. For example, the following set of functions allow the client application to construct a circuit step by step:

- `void` **`add_circuit`**`(`*`circuit_name`*`);`
- `void` **`add_gate`**`(`*`gate_type`*`, `*`gate_name`*`);`
- `void` **`add_line`**`(`*`source_gate`*`, `*`sink_gate`*`, `*`gate_port`*`);`
- `void` **`commit_circuit`**`();`

The idea behind this set of functions is that different client applications will usually operate on circuit specifications given in different formats (Verilog, VHDL, BLIF, etc.), and each client application needs to parse those specifications in any case. These functions can be called at the time of parsing with minimum overhead. When the client application has finished reading the circuit specification, also the Tiguan object will have built its internal circuit representation. As was explained in Section 4.4, Tiguan's internal representation of circuits is optimised such as to allow for the efficient formulation of SAT instances. For this reason, Tiguan needs to manage an own internal representation of the circuit.

Tiguan's interface also allows the client application to construct an n-time-frame expansion of the original circuit, and to switch between the original circuit and the expanded circuit at any time. The client application can address gates in the sequential expansion by the names that those gates have in the original circuit and by the identification number of the desired time frame, and the internal Tiguan manager takes care of mapping those addresses to the correct gates. In addition, functions are provided which allow the client application to query any type of data (numbers of inputs, outputs, gates, etc.) about the original or the expanded circuit at any time. For debugging purposes, the Tiguan library also offers functions to generate different types of graphical representations of the circuit and of Tiguan's internal processes.

Analogously to the construction of circuits, also a set of functions is provided which allows the client application to construct CMS@ (see Section 4.2) or ECMS@ (see Section 7.3) fault descriptions step by step:

- `void` **`add_new_fault`**`();`
- `void` **`add_aggressor`**`(`*`aggressor_gate`*`, `*`aggressor_type`*`);`

- `void add_victim(victim_gate, gate_port, stuck_at_type);`
- `void commit_fault();`

Also, the interface lets the client application add aggressors and victims in any order. This allows to collect the information that is required for the construction of fault descriptions in the way that is more suitable for the client application, and the appropriate Tiguan functions can be called on the fly with minimum disruption. In addition, several functions are provided with which the client application can operate on the fault list. For example, faults can be added and deleted at any time, and aggressors and victims can be added to existing fault descriptions. Furthermore, faults can be grouped into clusters (see Section 5.2), and the client application is able to form clusters according to its own criteria, and to process faults within clusters in any desired order.

Regarding the test generation process, the Tiguan interface allows the client application to customise a large number of parameters. Among other options, the client application can:

- set the simulation width for pattern-parallel fault simulation (see Section 4.5) or decide to not employ fault dropping;
- set timeouts or backtracking limits for the SAT solver;
- set the number of computation threads used by the SAT solver;
- set whether the SAT solver is to use pre-processing.

All these parameters can be set globally so that they will be valid for all calls of the **`generate_pattern`**-function, but they can also be overridden by custom values on a per-call basis. This allows to build client applications that process each fault using different parameters, for example for multi-stage processing (see Section 5.1.2).

Finally, the Tiguan interface also provides various functions for the management of the test set. For instance, the client application can decide whether a new generated test pattern shall be added to the test set, merged into the test set (see Sections 2.7.2 or 6.2), or discarded.

9.3 Grading of strong fault secureness

This section reports on a work published by Hunger et al. in [126]. The proposed approach reduces the estimation of a circuit's fault secureness to a test pattern generation problem where multiple stuck-at victims need to be considered at the same time, and where the outputs of some logic parts need to be restricted. Since such problems are easily formulated and efficiently solved using Tiguan's ECMS@-based test generation, Tiguan was used as back-end ATPG engine in this work.

The term "fault secureness" is a term used to characterise *self-checking* circuits. Let C be a self-checking combinational circuit, and let φ_C and φ_C^f denote the Boolean functions that are computed by C in absence of faults and in presence of a fault f, respectively. Also, let $\mathfrak{C}_{\text{in}}$ and $\mathfrak{C}_{\text{out}}$ be C's input code and output code, respectively[32].

In this context, the detectability of a fault f is defined only with respect to valid input patterns. That means that f is *detectable* if and only if there is an input pattern $p \in \mathfrak{C}_{\text{in}}$ with $\varphi_C^f(p) \neq \varphi_C(p)$. Conversely, f is *undetectable* if $\varphi_C^f(p) = \varphi_C(p)$ holds for all input patterns $p \in \mathfrak{C}_{\text{in}}$.

C is called *fault-secure* with respect to a fault f if and only if the following holds for all valid input patterns $p \in \mathfrak{C}_{\text{in}}$:

$$\text{If } \varphi_C^f(p) \neq \varphi_C(p), \text{ then } \varphi_C^f(p) \notin \mathfrak{C}_{\text{out}}.$$

This means that the checking logic can detect the presence of f if an output generated by C is not a valid code word. C is called *fault-secure* with respect to a fault list F if it is fault-secure with respect to all faults $f \in F$.

Note that fault secureness is trivially satisfied for undetectable faults. Such faults do not affect the circuit's functionality and are also not detected by the checking logic, but they can cause malfunction if further faults are present. In order to account for this, the notion of *strong fault secureness* was introduced in [232].

C is called *strongly fault-secure* with respect to a fault list F if and only if one of the following properties holds for each fault $f \in F$:

1. If f is detectable, C is *fault-secure* with respect to f.
2. If f is undetectable, then for all faults $f' \in F$ that can occur simultaneously with f, either Property 1 or Property 2 holds for the multiple fault $< f, f' >$.

[32] A *code* $\mathfrak{C}$ is defined as a set of words over the alphabet $\{0, 1\}$. Any word over $\{0, 1\}$ is called *valid* with respect to $\mathfrak{C}$ if and only if it belongs to $\mathfrak{C}$. Hence, the circuit's *input code* is the set of input patterns that are valid with respect to the used error-detecting or error-correcting coding scheme, while the circuit's *output code* is the set of valid output patterns.

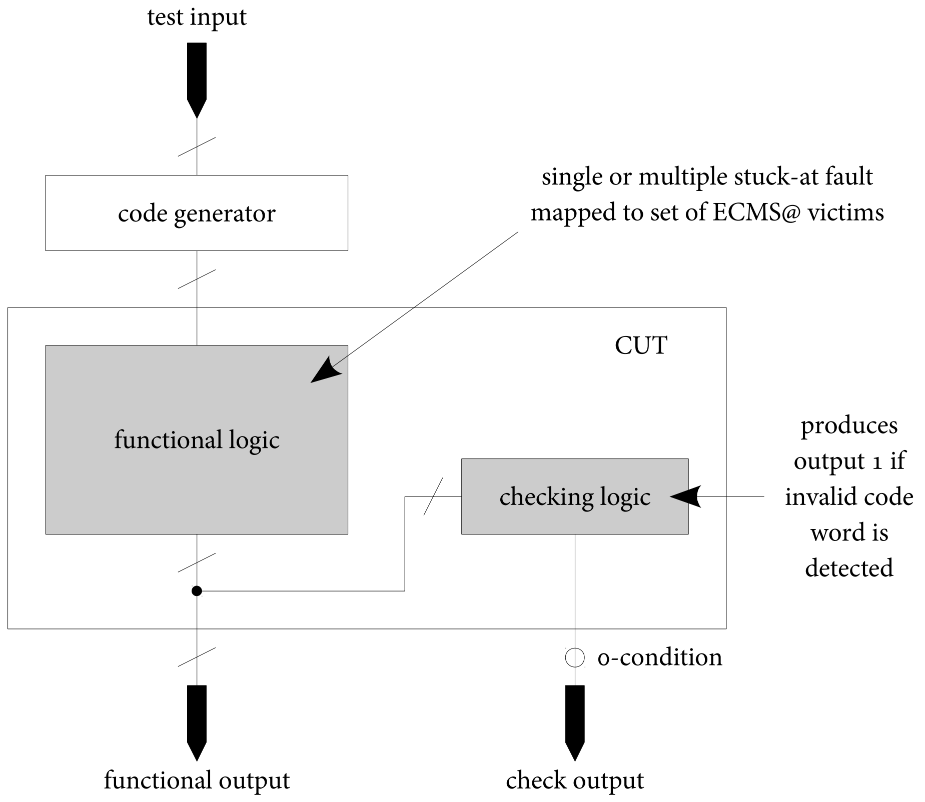


Figure 33. Test of a circuit's fault secureness — reduction to ECMS@-ATPG

In [125], Mr. Hunger presented a method to test a circuit's fault secureness with respect to a fault by reduction to ECMS@-ATPG. The principle of the reduction is explained in Figure 33. The circuit under test (CUT) includes the functional logic and the checking logic. For simplicity, the functional logic and the checking logic are shown as separate entities in Figure 33, but they can share components after the synthesis process. The checking logic reads the output pattern produced by the functional logic and produces the output 1 if and only if the output pattern is not a valid code word. The test bench also includes a code generator that restricts the set of inputs that can be applied to the CUT to valid input patterns. Alternatively, the conditions imposed by the code generator can also be integrated into the ECMS@ fault specification. In this work, however, the explicit modelling of the code generator's net list was preferred in order to enable the evaluation of various complex coding schemes.

For a given natural number $n \geq 1$ and n single-stuck-at faults $f_1, \ldots, f_n$ that can occur simultaneously, the method tests the CUT's fault secureness with respect to the multiple fault $<f_1,\ldots,f_n>$ by testing the detectability of an ECMS@ fault[33]. The complete test bench is passed to TIGUAN as a single combinational circuit. Each single-fault f_i within the functional logic is mapped to an ECMS@ victim line of appropriate type (s-a-0 or s-a-1), and a 0-condition is imposed on the output of the checking logic. Thus, the constructed ECMS@ fault can only be detected by test patterns under which the functional logic produces a wrong test response that is a valid output code word nonetheless. In consequence, the CUT is fault-secure with respect to the multiple fault $<f_1,\ldots,f_n>$ if and only if the constructed ECMS@ fault is undetectable.

In order to measure the strong fault secureness of a circuit C with respect to a fault list F, Mr. Hunger proposed in [126] a metric which is defined such as to reflect the circuit's tolerance to the accumulation of single faults. Hence, it is defined in function of each fault's *critical multiplicity*. For a single fault $f_0 \in F$, f_0's critical multiplicity $\kappa(f_0)$ is defined as follows[34]:

$$\kappa(f_0) := \begin{cases} 0 & \text{if } C \text{ is strongly fault-secure with respect to } f_0. \\ 1 & \text{if } C \text{ is not fault-secure with respect to } f_0. \\ \min\left\{ \kappa(<f_0,f_1,\ldots,f_n>) : \begin{array}{l} f_1,\ldots,f_n \in F \text{ such that } C \text{ is not fault-secure} \\ \text{with respect to } <f_0,f_1,\ldots,f_n>\text{, and such that} \\ <f_0,f_1,\ldots,f_i> \text{ is undetectable for all} \\ i = 0,\ldots,n-1. \end{array} \right\} & \text{else.} \end{cases}$$

The algorithm for the computation of the proposed metric works as follows: Starting with multiplicity 1, fault sequences of increasing multiplicity are analysed by reduction to ECMS@-ATPG as illustrated in Figure 33. For a multiplicity n, let F_n be the set of faults of that multiplicity that need to be analysed. At the beginning of the algorithm, F_n is set to F. (Single faults have multiplicity 1.) If the circuit is classified as strongly fault-secure with respect to a fault $<f_1,\ldots,f_n>\in F_n$ (i.e. the derived ECMS@ fault is undetectable), then $<f_1,\ldots,f_n>$ is removed from F_n and no additional fault accumulation is considered for that fault. Otherwise, the combination $<f_1,\ldots,f_n,f'>$ is added to F_{n+1} for all faults $f' \in F$ that can occur simultaneously with $<f_1,\ldots,f_n>$.

If the circuit is strongly fault-secure with respect to F, the algorithm eventually terminates when F_n becomes empty. In practice, however, the run-time depends strongly on the number of faults with higher multiplicity that need to be analysed, and this number can become very large after only few iterations. If the algorithm is

[33] For the ECMS@ fault, detectability is defined as usual, i.e. it is not restricted to an input code.

[34] The critical multiplicity κ is denoted by c in [126].

stopped at multiplicity N, the classification is not yet complete and the computed metric constitutes an optimistic estimation of the circuit's strong fault secureness. In order to reduce the number of multiple faults that need to be analysed explicitly by reduction to ATPG, several rules were derived which allow to analyse multiple faults based on the detectability of the single faults of which they are composed. Using these rules, the method was applied efficiently to ISCAS'85 benchmark circuits for $N = 2$. The experimental results are not presented here because their interpretation is out of the scope of this thesis. Details can be found in [126].

Recently, Viktor Fröse, doctoral student at Paderborn University, implemented an extension of this method that utilises TIGUAN's fault clustering technique in order to incrementally handle multiple faults with common prefixes. The aim of this extension is to increase the algorithm's run-time efficiency such that larger N-values can be reached, which improves the proposed metric's accuracy. The results obtained for this implementation have not been published yet, but speed-up factors of more than 100% were measured in preliminary experiments.

9.4 OPTIMISATION OF THE KLPG-ALGORITHM

Section 2.3.3 introduced different basic models for the test of defects that affect the timing behaviour of the circuit. The two most important models are the gate delay fault model (GDFM), which assumes that a single defective gate propagates either rising or falling transitions too slowly, and the path delay fault model (PDFM) which reflects reality better than the GDFM, as it models the accumulated effect of delay variations along a path. However, in the worst case, the number of paths in a circuit is exponential in the number of fan-out nodes. Hence, the generation of test patterns for every existing path is impractical.

For this reason, practical delay fault testing relies strongly on the efficient computation of a number of longest sensitisable paths in a given circuit. Although structurally longest paths are easily found by graph-traversal algorithms, the identification of sensitisable paths is significantly more challenging [257], as these paths must satisfy a large number of constraints depending on the robustness of the test pairs that are to be generated (see Section 2.4.3).

The efficient search for longest paths has been addressed by several authors, e.g. in [190, 222, 112, 45]. In [190], for instance, the well-known *KLPG-Algorithm* (K longest path generation) was presented, a structural algorithm that constructs K longest paths through each circuit gate. The procedure maintains a *path store* data structure that collects a number of partial paths and an estimate upper bound for the maximum length that those paths can reach upon completion. Those partial paths are continuously extended until they either become complete paths, or until their unsensitisability can be proved. In practice, the size of the path store data structure is restricted due to memory limitations. A positive effect of the size restriction is that the algorithm's run-time efficiency is improved, as the size restriction limits the size of the search space, but it can result in optimality loss. Due to the size limit of the path store, some partial paths can be excluded prematurely, i.e. before they have been completed or before their unsensitisability has been proved. Thus, actually longest sensitisable paths can be missed.

In [131], Jiang et al. presented an approach to improve the general KLPG-Algorithm introduced in [190]. Ms. Jiang implemented a KLPG-algorithm based on [190], and used this implementation to systematically evaluate the impact of limiting the maximum path store size $\pi_{\max}$ on the optimality of the results. Moreover, the cost of generating an optimal solution was investigated, and a provably optimal algorithm named *Opt-KLPG* was proposed. Opt-KLPG outperforms the conventional KLPG-Algorithm which requires larger path stores to produce better solutions.

Next, a brief description of the implemented algorithm follows. Details can be found in [131]. Given a set of targeted gates, the algorithm searches for the K longest sensitisable paths that pass through each targeted gate. For each target gate, the path store is progressively filled with the partial paths that start at a primary input in the targeted gate's input cone. The partial paths are extended gate by gate until K sensitisable paths have been found, or until the path store becomes empty. In order to extend a path, the off-path inputs of the gate by which the path is extended are assigned logic values according to the required sensitisation condition (robust, non-robust, etc., see Section 2.4.3), and the implications of those assignments are calculated. If an assignment results in a local conflict, the partial path is excluded from further consideration. For each partial path under consideration, a value called *max esperance* is computed and continuously updated. It represents the upper bound for the length that the partial path can reach upon completion, and it determines which partial paths are selected first for extension. Whenever a path of sufficient length and without local conflicts is completed, the algorithm tests its global sensitisability. For this purpose, the algorithm constructs a CMS@ fault employing the interface introduced in Section 9.2. For each value that needs to be assigned to the off-path input of a gate on the candidate path, an aggressor of appropriate type is added to the fault description. The only victim added to the fault description corresponds to a stuck-at fault located at the output of the targeted gate in the second time frame. Note that the circuit and its sequential expansion need to be instantiated only once. Once the TIGUAN object has been initialised and a circuit has been set up, any number of fault descriptions can be constructed and solved in any order without the need to reinitialise the TIGUAN object. Thus, the communication overhead between the KLPG-Algorithm and the TIGUAN engine is reduced. If TIGUAN finds a test that detects the constructed CMS@ fault, that test also sensitises the tested path. If TIGUAN reports that the CMS@ fault is undetectable, the path is unsensitisable.

The proposed Opt-KLPG-Algorithm corresponds to an iterative version of this basic algorithm. It starts by running KLPG using a path store with a fixed capacity $\pi_{\max}$. The first iteration of Opt-KLPG yields a set of sensitisable paths through each targeted gate. However, this set of sensitisable paths may be sub-optimal with respect to the length of the found paths. In contrast to KLPG, however, Opt-KLPG records the partial paths that cannot be kept in the path store due to its capacity limit (*overflow paths*). If a target gate is affected by an overflow, subsequent iterations of Opt-KLPG invoke further KLPG runs (using the same $\pi_{\max}$-value). In these iterations, the path store is initialised with the overflow paths generated in previous iterations. The procedure is repeated until all overflow paths have been processed. Thus, the solution produced by Opt-KLPG is always optimal for any $\pi_{\max}$-value. However, $\pi_{\max}$ has an effect on the algorithm's run-time efficiency.

An example is shown in Table 35. This corresponds to one ISCAS'85 circuit and one NXP circuit out of eight circuits reported on in Table I in [131]. The second column quotes the $\pi_{\max}$-value used in the corresponding row. The next five columns show the results obtained by KLPG, and the last six columns show the results obtained by Opt-KLPG. The columns labelled *overflows* quote the number of partial paths that were dropped due to path store overflow. For Opt-KLPG, this number is always higher because KLPG constitutes only the first iteration of Opt-KLPG. For Opt-KLPG, Column *rep* quotes the number of repetitions, i.e. the number of iterations that were required after the initial KLPG run until no more overflows occurred. The quoted values are the averages over all target gates. The columns labelled *length* show the sum of the lengths of the identified paths. In the case of KLPG, this number is optimal only if the number of overflows is 0. For Opt-KLPG, this number is always optimal.

The last three columns of each fault group quote the run-time of the respective approach in seconds, partitioned into the time needed for the actual path search (columns labelled *srch*) and the time consumed by all runs of TIGUAN in order to check the sensitisability of candidate paths (columns labelled *chck*). The columns labelled *total* quote the sum of these two numbers. For both circuits and all $\pi_{\max}$-values less than or equal to 3000, the total run-time of KLPG lies below the run-time of Opt-KLPG, which is owed to the additional iterations executed by Opt-KLPG.

Table 35
Search for longest sensitisable paths with Opt-KLPG (taken from [131])

		KLPG					Opt-KLPG					
				run-time (s)						run-time (s)		
circuit	$\pi_{\max}$	overflows	length	srch	chck	total	overflows	rep	length	srch	chck	total
c1908	10	143,619	75,569	4	32	36	332,249	42.25	86,185	26	320	346
	50	215,713	82,819	8	68	76	280,743	7.56	86,185	22	315	337
	100	125,803	85,717	8	60	68	163,067	2.46	86,185	21	288	309
	500	13,356	86,155	13	69	82	29,950	0.08	86,185	42	264	306
	1,000	14,932	86,181	30	92	122	23,696	0.03	86,185	77	266	343
	1,500	16,278	86,181	56	121	177	20,164	0.02	86,185	122	264	386
	3,000	12,902	86,181	186	189	375	12,902	0.01	86,185	252	274	526
	30,000	0	86,185	510	305	815	0	0.00	86,185	490	284	774
p78k	10	166,206	6,342	101	139	240	689,810	1.20	7,149	418	6,721	7,139
	50	177,257	7,039	112	279	391	501,469	0.18	7,149	198	5,253	5,451
	100	99,069	7,147	108	227	335	373,759	0.07	7,149	174	5,309	5,483
	500	135,763	7,148	125	578	703	321,940	0.01	7,149	245	4,995	5,240
	1,000	150,340	7,148	180	988	1,168	299,284	0.01	7,149	413	6,649	7,062
	1,500	151,226	7,149	208	1,136	1,344	283,626	0.00	7,149	498	4,601	5,099
	3,000	148,676	7,149	461	1,796	2,257	243,633	0.00	7,149	927	4,618	5,545
	30,000	0	7,149	5,859	4,731	10,590	0	0.00	7,149	5,840	4,720	10,560

However, KLPG's solution is sub-optimal in all of these cases. KLPG is able to return the optimal solution only for $\pi_{\max}$ =30,000, and in this case, KLPG's run-time is larger than the run-time needed by Opt-KLPG using any path store size. Thus, Opt-KLPG is superior, in particular when memory limitations need to be observed.

Regarding the performance of the SAT-based checking of the sensitisability of candidate paths in Opt-KLPG, it can be seen that the total time consumed by all Tiguan-calls (*chck*) tends to decrease for larger $\pi_{\max}$-values, as the Tiguan-procedure needs to be called less often because the necessity for re-iterations decreases. But, in general, the SAT-based checking of candidate paths accounts for a large fraction of the total run-time. However, it must be remarked that Tiguan's incremental capabilities (e.g. the fault clustering technique, see Section 5.2) were not employed in this application. Given that Opt-KLPG constructs large sets of CMS@ faults that are very similar (corresponding to complete paths with many common partial sub-paths, especially in the re-iterations), this is an application that will probably benefit very strongly of Tiguan's most advanced speed-up techniques. Hence, an interesting direction for future research consists in the incorporation of those techniques into this promising application.

9.5 Conclusions

The aim of this chapter was to provide the reader with an idea of how vast the application possibilities of SAT-based ATPG are. The SAT-ATPG tool TIGUAN, which was introduced in Chapter 4, and whose numerous enhancements have been the subject of the subsequent chapters, was made available to other researchers in the form of a C++ library that allowed to incorporate TIGUAN's functionality into diverse client applications. The interface was designed such as to give the client applications maximum flexibility.

In order to illustrate the applicability of the library, the rest of the chapter summarised two works that made use of this library. The first work (Section 9.3), which consisted in evaluating a self-checking circuit's tolerance to the accumulation of multiple faults by reduction to ECMS@-based SAT-ATPG, constitutes an especially interesting application of TIGUAN, because the topic of fault tolerance as a measure to counter process variations is expected to continue gaining in importance. The defect behaviour studied in these areas is characterised by high variability. This makes it necessary to either address the same targets multiple times using different parameters, or to formulate problems in a flexible way that allows to specify preferred solutions. TIGUAN has been shown to handle these two cases with great efficacy, e.g. using the fault clustering technique (see Chapter 5), or using SAT solving with preferences (see Chapter 7).

10

Summary and concluding remarks

The application of the exhaustive test set composed of 2^n test patterns to a combinational n-input circuit is factually impermissible for any circuit of practical interest. 2^n amounts to 65,536 for an n-value as low as 16, and to 4,294,967,296 for $n = 32$ (the number of inputs of a 16-bit adder). Thus, the dedicated computation of specific test patterns is the most important task in hardware testing. Without powerful ATPG algorithms able to process large numbers of modelled faults with feasible computational effort, hardware test would be impossible altogether.

Although the reduction of ATPG to the problem of Boolean satisfiability was proposed as far back as 1968, SAT-based ATPG remained of purely academical interest until fairly recently. Relevant optimisations of the basic algorithm were proposed in the beginning of the 1990s, but the run-time efficiency achieved by structural algorithms was still ahead of SAT-based approaches of that time.

However, research on efficient SAT solving made after the year 2000 changed the landscape. Driven by problems in the field of formal verification, where the typical workload is characterised by few, but very hard and usually unsatisfiable SAT instances, SAT solving research engendered techniques that allow modern SAT solvers to prune large parts of the search space very efficiently. Transferred to test pattern generation, this meant that SAT-based ATPG was now able to process undetectable and especially hard-to-detect faults more efficiently than structural ATPG, and this lead to a resurgence of the study of SAT-based ATPG.

This doctoral thesis revolved around the efficient implementation and the improvement of various aspects of the SAT-based ATPG tool Tiguan (**T**hread-parallel **I**ntegrated test pattern **G**enerator **U**tilising satisfiability **AN**alysis). Starting with the basic principles of Tiguan's core algorithms, the first half of the thesis discussed in detail different techniques that significantly improved Tiguan's run-time efficiency.

But run-time efficiency is not the only criterion to evaluate the quality of test pattern generation tools. Due to the high cost that test application time represents to the semiconductor industry, and due to the memory limitations of automatic test equipment, also the ATPG tool's capability to produce compact test sets is a relevant indicator. Thus, also the compaction capability of Tiguan was systematically studied and improved until turning SAT-ATPG into an attractive alternative to structural approaches.

The second half of the thesis went even further and showed that the potential of SAT-ATPG goes beyond being an alternative or a complement of structural methods in the test generation for standard fault models. The flexibility offered by the basic Tiguan framework was combined with newest advances in SAT solving in order to venture into the solution of realistic, defect-based ATPG problems that require the satisfaction of additional optimisation goals. In general, such problems cannot be solved optimally using structural approaches. Thus, the versatile, generic fault models defined in this thesis, and the integration of their support into Tiguan, constitute a significant, non-trivial contribution to the area of test pattern generation. After the demonstration of this powerful framework's range of capabilities by means of an example application, also various applications relevant in the context of process variations and fault tolerance were discussed.

In more detail, Chapter 4 introduced the SAT-based test pattern generator Tiguan and explained in detail all aspects that were considered in order to allow for an especially efficient implementation. From the beginning, Tiguan was crafted around a new, generic fault model — the conditional multiple stuck-at fault model (CMS@FM), which allows to model defects with fault effects on multiple victim lines that are activated simultaneously if a number of aggressor lines satisfy certain conditions. The first application introduced in order to illustrate the possibilities offered by the CMS@FM was the test generation for gate-exhaustive testing, an approach that tests each stuck-at fault under all input combinations that can be applied to the gate driving the fault location in order to sensitise the fault.

The experiments discussed in Chapter 4 showed that Tiguan was able to classify all single-stuck-at faults in large industrial circuits provided by NXP Semiconductors GmbH Hamburg. Moreover, Tiguan outperformed not only the tool PASSAT developed at the University of Bremen, but also a commercial tool that uses a structural algorithm; the latter was not able to classify all faults in nxp circuits, even using a high conflict limit. In particular, the obtained results showed that SAT-based ATPG performs especially well for faults that are too hard for structural algorithms. Although structural algorithms continue to display better run-times for the majority of easy-to-detect faults, this proves that SAT-based ATPG is indispensable in order to obtain full fault coverage.

Chapter 5 was dedicated to the study of techniques that allow to improve the general run-time efficiency of SAT-based ATPG. The first part of the chapter concentrated on how to speed up SAT-ATPG based on the best utilisation of the SAT solving engine. First the most relevant details of the SAT solver MiraXT's core algorithms were introduced, and special attention was paid to customisations implemented into the SAT solver in order to achieve the best SAT solving performance for the type of SAT instances generated by Tiguan. In addition, the scalability of Tiguan to multi-core systems was evaluated, as MiraXT is able to distribute the SAT solving process among multiple threads that can be executed in parallel. It was determined that Tiguan benefits from thread-parallel SAT solving on systems with around 4–6 available cores, while larger numbers of cores do not lead to further speed-up. The reasons for this phenomenon were analysed in order to derive directions for future research. Furthermore, the study discovered that thread parallelism is mostly ineffective for the processing of easy-to-solve ATPG instances, while very hard instances benefit strongly from increased resources. In consequence, a two-stage technique that employs thread parallelism only for the processing of harder faults was implemented and successfully applied to stuck-at faults and to CMS@ faults for gate-exhaustive testing.

The second part of Chapter 5 introduced the SAT engine antom and explained the most important differences between antom and MiraXT from the point of view of a SAT-ATPG application. An important observation made in Chapter 4 was that the time needed for the formulation of SAT instances is not only non-negligible, but on average even higher than the time needed for SAT solving. Motivated by this, a fault clustering technique was implemented, which reduces the number of SAT generation runs significantly, while the SAT solving runs benefit from antom's incremental SAT solving. Finally, various criteria for the grouping of faults into clusters were evaluated. The best strategy allowed Tiguan to classify all stuck-at faults in large industrial circuits in 47.7% less run-time.

After the discussion of techniques to enhance the run-time efficiency of Tiguan, test set compactness was addressed in Chapter 6. A dynamic compaction procedure was introduced, which was specially designed for integration into a SAT-based framework, and which makes use of Tiguan's specific data structures and interfaces. The technique was found to scale also to large industrial designs, and it outperformed static compaction both in terms of test compactness and run-time. Furthermore, the dynamic compaction procedure enabled Tiguan to generate more compact test sets for iscas circuits than a commercial ATPG tool that implements a structural algorithm. Regarding the application of the proposed technique to NXP circuits, the test sets produced by the commercial tool are still more compact than those produced by Tiguan, but the gap between structural and SAT-based ATPG in terms of test compactness was significantly diminished.

From Chapter 7 onwards, the thesis concentrated on the application of SAT-ATPG to complex fault models. Although the stuck-at FM has been the dominant fault model used in practical applications for a long time, it has been shown beyond doubt that it does not accurately reflect several defect types encountered in currently employed technologies. The first part of this chapter discussed the application of CMS@-based SAT-ATPG to generate test patterns for bridging faults with non-zero resistance, which are a very good example of a realistic defect model that requires the handling of multiple stuck-at victims. The second part of the chapter introduced an extension of the SAT-ATPG framework around the CMS@FM. The most important feature of the new model — the enhanced conditional multiple-stuck-at fault model (ECMS@FM) — is that it provides an easy-to-use method to specify optimisation constraints that guide the SAT-ATPG process towards the generation of preferred patterns. The implementation of ECMS@ support was explained in detail and several applications were discussed. For instance, test patterns were generated which sensitise the maximum achievable number of primary outputs, as well as test patterns that provide precise control of local switching activity. These applications exemplify the type of non-trivial optimisation problems that are very difficult to solve using tools based on structural ATPG.

An important aspect of ECMS@-based SAT-ATPG is that the generated solutions are always optimal with respect to the chosen problem formulation at ECMS@ level. However, the way in which the mapping from the original problem to the ECMS@FM is done has a large effect on the accuracy of the solutions and on the algorithm's run-time efficiency. This problem was explored in Chapter 8, where ECMS@-based SAT-ATPG was used to generate test patterns for power droop testing. ATPG for power droop constitutes an extremely hard variation of sequential test generation, given that a large number of times frames need to be modelled and that three different optimisation objectives need to be satisfied simultaneously. The experimental results showed the applicability of the method to mid-sized benchmark circuits. In particular, the analysis performed in this chapter demonstrated the relevance of intelligent mapping from the original problem to ECMS@-ATPG, and called attention to important subjects of interest in future research.

Finally, Chapter 9 provided insight into further application possibilities for SAT-based ATPG. This chapter introduced the interface of the Tiguan library, a C++ library that was developed in order to allow client applications to incorporate Tiguan's functionality. The interface was designed with two main goals in mind: maximum flexibility for the client application and efficient communication between the client application and the SAT-ATPG back-end. One of the discussed examples is an approach developed at Paderborn University, where the evaluation of a self-checking circuit's tolerance to the accumulation of multiple faults was reduced to ECMS@-based SAT-ATPG. This demonstrated the applicability of SAT-ATPG to problems that are relevant in the context of process variations and fault tolerance. These topics are expected to continue gaining importance, and they often require to either address the same targets multiple times using different parameters, or to formulate problems in a flexible way that allows to specify preferred solutions. Throughout this thesis, it was shown that TIGUAN is especially well-suited for such tasks.

In summary, the most important contributions of the work presented in this thesis are as follows:

- The general run-time efficiency of SAT-ATPG was increased through intelligent mapping of the ATPG problem to SAT, through the optimal utilisation of multiple computing cores, and through the employment of advanced SAT solving techniques.
- Dynamic compaction was integrated into SAT-ATPG. The application of the method to ISCAS and ITC'99 circuits resulted in smaller test sets than those produced by a commercial, structural tool. For industrial circuits, the compaction efficiency gap between SAT-based and structural ATPG was significantly diminished.
- Generic fault models were defined which allow to represent complex defect behaviour. In addition, a flexible SAT-based framework for the generation of provably optimal test patterns for complex fault models was implemented. The applicability of the framework was illustrated by several example applications whose replication using structural methods is not trivial.
- The performed work opened the path to advanced research in small-delay test, variability and fault tolerance. A well-documented C++ library with a multifunctional interface was provided for the incorporation of the developed SAT-ATPG methods into client applications. In addition, the expertise gained by the author in the field of SAT-ATPG and the created software code base served as starting point for numerous SAT-based methods developed by fellow doctoral students in Freiburg [208, 206, 209, 211, 210, 207, 52].

Given all these advancements in the field of SAT-based ATPG, one important question remains to be discussed, namely whether SAT-based techniques will eventually be integrated into commercial ATPG tools. The level of attention that industry attendees have paid to works on SAT-ATPG presented at international symposia in recent years has been very high, which means that the potential of SAT-based methods has been recognised.

Most importantly, it has been shown beyond doubt that SAT-based ATPG outperforms structural methods regarding the application to undetectable and hard-to-detect stuck-at faults. However, structural ATPG continues to display a better run-time performance for the majority of easy-to-detect faults. It is probable that industry researchers have been attempting to enhance existing structural ATPG algorithms by importing some of the strategies and principles employed in SAT-based ATPG. But the point of view of the author is that that is possible only up to a limited extent, because SAT-based and structural methods have different properties. Much of the efficiency of SAT solving stems from the fact that clauses are very simple and homogeneous data structures, and that a SAT formula can be represented internally as a low-cost queue of clauses. In contrast, reasoning based on a gate-level net list requires the implementation of data structures for graph traversal algorithms. Hence, the SAT domain is significantly more efficient in deriving the implications of decisions made during the search process.

Going into more detail, the search tree of a structural ATPG algorithm, e.g. the D-Algorithm, is different from a SAT solving search tree. Since each line is represented by several Boolean variables, in SAT there are more variables, and the SAT search tree is larger; but it is more uniform because all nodes are of the same type. In the D-Algorithm's search tree, in contrast, each node has a type (the gate type) that determines what kind of implications can be derived at that specific node. Also, the D-Algorithm needs to observe more constraints that do not exist in the SAT domain, e.g. whether propagation shall be given precedence over justification or vice versa. Altogether, the uniformity of the SAT search tree has given rise to easy-to-apply, provably correct inference rules that allow to identify large sub-spaces that can be excluded from the search process. SAT is often able to infer implications that may refer to structurally distant points in the net list, i.e. implications that are not easily identifiable using the D-Algorithm.

This is the main reason why SAT-based ATPG is better suited to prove a fault's unsatisfiability, or to process hard-to-detect faults. In contrast, the D-Algorithm usually needs to make a large number of decisions in order to prove undetectability, and in consequence, the cost of backtracking is accumulated during the search.

In summary, it can be said that structural ATPG performs best on instances in which the amount of conflicts is lower. The attempt to incorporate SAT-specific learning techniques into structural algorithms would inevitably slow down their performance on easy-to-detect faults. Reasoning in SAT is based on the collection and management of a large amount of learnt clauses, which can outnumber the clauses in the original SAT instance significantly. For structural algorithms, the implementation and maintenance of such a learning infrastructure would result in a computational effort that is not justified for most ATPG instances.

An alternative that has been proposed is the combination of structural and SAT-based ATPG, where both types of methods are applied to different faults [242]. This solution, however, results in the need to implement and maintain two different ATPG engines, the cost of which could be a factor delaying the industry's eventual adoption of SAT-based techniques.

For industrial applications, another critical issue is the over-specification of patterns generated using SAT-based methods. In this thesis, a dynamic compaction method was presented which overcomes this shortcoming by extracting necessary detection conditions not from the values assigned to the primary inputs, but from internal circuit lines. The presented compaction technique was successful in constructing compact test sets, but test patterns with high densities of unspecified bits are needed also for other purposes; for instance, for test data compression. In consequence, an important direction for future research on SAT-ATPG will be the incorporation of efficient pattern relaxation techniques (see also Section 6.5).

Still, in the author's opinion, industry may have to resort to SAT-based methods in the long run. Not as a substitute for structural algorithms, but at least as a complement — SAT-based ATPG can handle highly constrained problems more efficiently. And increasingly, SAT-ATPG may become indispensable to manage different kinds of problems that require complex fault modelling. Chapters 7 and 8 of this thesis discussed several applications in which realistic, defect-based modelling made it necessary to impose a large number of conditions on aggressor or observation lines, to dynamically control the paths through which fault effects are propagated, and to satisfy optimisation criteria, whereas the solution of such problems using structural methods is more difficult.

And even more complex modelling approaches can be expected to emerge in the future. An interesting example is the *robust enhanced aggressor-victim* model (REAV) proposed by Hillebrecht et al. in [118] to emulate the behaviour of interconnect opens. The main challenge of realistic modelling of interconnect open defects is that the actual behaviour of the gate that is driven by the open line depends on physical parameters that are usually unknown, like the amount of charge trapped in the disconnected segment of the line. The model needs to take the analog behaviour of the involved gates into account, albeit without increasing the complexity of the ATPG problem to a computationally unfeasible level [116].

In the REAV model, the open line is the victim, while a set of neighbouring lines are seen as aggressors. Due to the coupling capacitance between the victim line and the aggressor lines, the exact voltage on the victim will vary depending on what values are justified on the aggressor lines. The gate driven by the victim line will behave correctly or erroneously depending on the exact voltage measured on the victim. In contrast to the test generation for resistive-bridging faults (see Sections 2.5 and 7.2), where the values applied to the inputs of the gates driving the bridged lines are fixed for each tested section, in this work, the number of aggressors can be very large. Therefore, fixing all their values in advance can result in a large fraction of unsatisfiable ATPG instances. Moreover, it is not necessary to fix the value of all aggressors. Different aggressor assignments will usually result in different voltages on the victim line, but for the purpose of fault detection, it suffices to justify any aggressor assignment that induces a voltage on the victim above or below certain thresholds. Hence, this is an ATPG problem that accepts various assignment combinations made to the aggressors.

In Chapter 7, it was shown that it is possible to integrate conditions that affect the behaviour of a number of lines into the SAT-ATPG problem (e.g. maximise the number of observed lines that switch), and that optimal solutions can be guaranteed using ECMS@-based SAT-ATPG, whereas the computation of optimal solutions using structural algorithms is more difficult. Therefore, ECMS@-ATPG is more suitable for an application such as the test generation for REAV-modelled open defects. The REAV-model, however, necessitates the satisfaction of conditions that depend on real-valued parameters, like the coupling capacitance between the victim and the aggressors, and the voltage thresholds. Thus, a further extension of ECMS@-ATPG in order to allow it to handle such conditions in an efficient way constitutes a very important topic for future research.

Aside from further topics for future research discussed in detail in the concluding section of each chapter of this thesis, SAT-based diagnostic ATPG[35] is also an interesting application. Recently, a Tiguan-based prototype was implemented by an undergraduate student under the advice of the author of this thesis. The potential advantages of ECMS@-based diagnostic ATPG are multiple. For example, the internal representation of defects as ECMS@ faults allows to explicitly distinguish between faults corresponding to different fault models or between arbitrary defects whose behaviour can be properly mapped to the ECMS@FM. Furthermore,

[35] *Diagnostic ATPG* has the aim of constructing test patterns that distinguish faults. Two faults f_1 and f_2 are *distinguished* by a test pattern p if $\varphi_C^{f_1}(p) \neq \varphi_C^{f_2}(p)$, i.e. if the response to p's application is different under the influence of f_1 and f_2.

the ability of ECMS@-ATPG to control the behaviour of observation lines can be used to achieve a number of different goals in diagnostic ATPG; for instance, the minimisation of the number of affected outputs in order to increase the diagnostic resolution. In addition, diagnostic ATPG can also be combined with TIGUAN's fault clustering technique such that certain sets of fault pairs to be distinguished can be processed incrementally. A systematic exploration of the possibilities offered by such a combination will lead to improve the quality of diagnostic test patterns.

As can be seen, the methods and techniques presented in this thesis have gone far beyond enhancing the run-time efficiency and compaction abilities of SAT-based ATPG for stuck-at faults. The definition of powerful and flexible, generic fault models, paired with the handling of optimisation goals, has given rise to efficient, defect-based test generation using SAT, and it has opened the door to a myriad of relevant applications in a time in which complex fault models are no longer a concept of mere academic interest. It was already pointed out in [179] that academic developments typically require several years to be integrated into commercial CAD software, but in the long run, industry will likely start embracing SAT-based methods due to the indisputable reasoning power they bear. And then, feedback from industry will surely lead to even more interesting problems.

Appendix A

Benchmark details

This appendix presents detailed information on the four suites of benchmark circuits that were employed in the experiments reported on in this thesis. Each table contains the circuit names employed throughout this thesis, along with the circuits' number of gates, depth, number of primary inputs and number of primary outputs. In the case of sequential circuits, also the number of flip-flops is quoted.

Table 36
ISCAS'85 benchmark circuits [32]

circuit	gates	depth	inputs	outputs
c0017	13	5	5	2
c0095	39	6	5	7
c0432	203	19	36	7
c0499	275	13	41	32
c0880	469	26	60	26
c1355	619	26	41	32
c1908	938	42	33	25
c2670	1,566	34	233	140
c3540	1,741	49	50	22
c5315	2,608	51	178	123
c6288	2,480	126	32	32
c7552	3,827	45	207	108

Table 37

ISCAS’89 BENCHMARK CIRCUITS [31]

circuit	gates	depth	inputs	outputs	flip-flops
s00027	21	8	4	1	3
s00208	131	13	10	1	8
s00298	156	11	3	6	14
s00344	210	22	9	11	15
s00349	211	22	9	11	15
s00382	209	11	3	6	21
s00386	185	13	7	7	6
s00400	213	11	3	6	21
s00420	269	15	18	1	16
s00444	232	13	3	6	21
s00510	249	14	19	7	6
s00526	245	11	3	6	21
s00641	476	76	35	24	19
s00713	489	76	35	23	19
s00820	336	12	18	19	5
s00832	334	12	18	19	5
s00838	545	19	34	1	32
s00953	492	18	16	23	29
s01196	593	26	14	14	18
s01238	572	24	14	14	18
s01423	827	61	17	5	74
s01488	692	19	8	19	6
s01494	686	19	8	19	6
s05378	3,221	27	35	49	179
s09234	6,094	60	36	39	211
s13207	9,441	61	62	152	638
s15850	11,067	84	77	150	534
s35932	19,876	31	35	320	1,728
s38417	25,585	49	28	106	1,636
s38584	22,447	58	38	304	1,426

The first three suites are widely used in academia. Details on the combinational ISCAS’85 circuits are listed in Table 36.

Details on the ISCAS’89 suite are given in Table 37. These circuits are sequential. In experiments in which an algorithm for combinational circuits was applied to the combinational cores of these circuits, the circuits are denoted by the names listed in Table 37 preceded by *c*. For example, the combinational core of circuit s00027 is denoted by *cs00027*.

Table 38

ITC'99 BENCHMARK CIRCUITS [7, 48]

circuit	gates	depth	inputs	outputs	flip-flops
b01	54	8	2	2	5
b02	31	7	1	1	4
b03	183	12	4	4	30
b04	694	34	11	8	66
b05	608	39	1	36	34
b06	64	7	2	6	9
b07	476	33	1	8	49
b08	192	14	9	4	21
b09	188	11	1	1	28
b10	197	14	11	6	17
b11	579	39	7	6	31
b12	1,127	21	5	6	121
b13	370	13	10	10	53
b14	5,923	43	32	54	245
b15	8,026	47	36	70	449
b17	25,719	46	37	97	1,414
b18	76,513	92	37	23	3,270
b20	12,991	75	32	22	490
b21	13,168	75	32	22	490
b22	18,789	80	32	22	703

The ITC'99 suite is a further set of well-known sequential circuits. Their details are quoted in Table 38. In experiments in which an algorithm for combinational circuits was applied to the combinational cores of these circuits, the circuits are denoted by the names listed in Table 38 succeeded by *c*. For example, the combinational core of circuit b01 is denoted by *b01c*.

Table 39
Industrial benchmark circuits provided by NXP Hamburg [5]

circuit	gates	depth	inputs	outputs	remarks
p35k	48,927	68	2,912	2,229	p44k
p45k	46,075	57	3,739	2,550	–
p77k	75,033	568	3,487	3,400	–
p78k	80,875	48	3,148	3,484	–
p81k	96,722	57	4,029	3,952	p80k
p89k	92,706	110	4,683	4,557	p88k, tri-state
p100k	102,443	102	5,902	5,829	p99k
p141k	185,360	75	11,290	10,502	p177k, tri-state
p267k	296,404	72	17,332	16,621	–
p269k	297,497	72	17,333	16,621	tri-state
p286k	373,221	127	18,411	17,827	tri-state
p295k	311,901	116	18,508	18,521	tri-state
p330k	365,492	73	18,010	17,468	–
p378k	404,367	48	15,732	17,420	–
p388k	506,034	224	25,005	24,065	tri-state
p469k	49,771	128	635	403	p49k
p951k	1,147,491	135	92,027	104,747	p1330k, tri-state
p1522k	1,193,824	515	71,414	68,035	tri-state
p2927k	2,539,052	400	101,844	95,143	–

Finally, also a set of nineteen large, industrial circuits containing up to two and a half million gates were employed. These circuits were provided by NXP Semiconductors GmbH Hamburg [5] and are referred to as NXP circuits throughout this thesis. These circuits are sequential, but since only their combinational cores were used in all experiments, the original number of flip-flops is not listed in Table 39. The last column of this table (*remarks*) gives additional information on some of these circuits, for instance whether they contain tri-state elements.

Also, the thesis compares the performance of Tiguan to a SAT-ATPG tool developed by a research group at the University of Bremen (see Section 4.6). However, the name employed in Bremen to refer to some of these circuits is different. Where that is the case, the name used in Bremen is listed in column *remarks*.

Author's publications

Journal articles

[J1] M. Sauer, A. Czutro, T. Schubert, S. Hillebrecht, I. Polian and B. Becker, "SAT-based Analysis of Sensitisable Paths," *IEEE Design & Test of Computers*, 2013.

[J2] A. Czutro, I. Polian, M. Lewis, P. Engelke, S.M. Reddy and B. Becker, "Thread-Parallel Integrated Test Pattern Generator Utilizing Satisfiability Analysis," *International Journal of Parallel Programming*, vol. 38, pp. 185–202, June 2010.

[J3] I. Polian, A. Czutro, S. Kundu and B. Becker, "Power Droop Testing," *IEEE Design & Test of Computers*, vol. 24, pp. 276–284, May 2007.

Papers in formal proceedings (refereed)

[C1] A. Czutro, M. Imhof, J. Jiang, A. Mumtaz, M. Sauer, B. Becker, I. Polian and H.-J. Wunderlich, "Variation-Aware Fault Grading," in *IEEE Asian Test Symposium*, pp. 344–349, November 2012.

[C2] M. Sauer, A. Czutro, I. Polian and B. Becker, "Small-Delay-Fault ATPG with Waveform Accuracy," in *International Conference on Computer-Aided Design*, pp. 30–36, November 2012.

[C3] M. Sauer, S. Kupferschmid, A. Czutro, I. Polian, S.M. Reddy and B. Becker, "Functional Test of Small-Delay Faults using SAT and Craig Interpolation," in *International Test Conference*, pp. 1–8, November 2012.

[C4] L. Feiten, M. Sauer, T. Schubert, A. Czutro, E. Böhl, I. Polian and B. Becker, "#SAT-Based Vulnerability Analysis of Security Components — A Case Study," in *International Symposium on Defect and Fault Tolerance*, October 2012.

[C5] A. Czutro, M. Sauer, I. Polian and B. Becker, "Multi-Conditional SAT-ATPG for Power-Droop Testing," in *IEEE European Test Symposium*, May 2012.

[C6] M. Sauer, A. Czutro, B. Becker and I. Polian, "On the Quality of Test Vectors for Post-Silicon Characterization," in *IEEE European Test Symposium*, May 2012.

[C7] A. Czutro, M. Sauer, T. Schubert, I. Polian and B. Becker, "SAT-ATPG Using Preferences for Improved Detection of Complex Defect Mechanisms," in *VLSI Test Symposium*, April 2012.

[C8] J. Jiang, M. Sauer, A. Czutro, B. Becker and I. Polian, "On the Optimality of K Longest Path Generation Algorithm Under Memory Constraints," in *Conference on Design, Automation and Test in Europe*, pp. 418–423, March 2012.

[C9] M. Sauer, S. Kupferschmid, A. Czutro, S.M. Reddy and B. Becker, "Analysis of Reachable Sensitisable Paths in Sequential Circuits with SAT and Craig Interpolation," in *International Conference on VLSI Design*, January 2012.

[C10] M. Sauer, J. Jiang, A. Czutro, I. Polian and B. Becker, "Efficient SAT-Based Search for Longest Sensitisable Paths," in *IEEE Asian Test Symposium*, November 2011.

[C11] M. Sauer, A. Czutro, I. Polian and B. Becker, "Estimation of Component Criticality in Early Design Steps," in *IEEE International On-line Testing Symposium*, pp. 104–110, July 2011.

[C12] M. Sauer, A. Czutro, T. Schubert, S. Hillebrecht, I. Polian and B. Becker, "SAT-Based Analysis of Sensitisable Paths," in *IEEE Design and Diagnostics of Electronic Circuits and Systems*, pp. 93–98, April 2011. Best Paper Award in the Test Category.

[C13] A. Czutro, I. Polian, P. Engelke, S.M. Reddy and B. Becker, "Dynamic Compaction in SAT-Based ATPG," in *IEEE Asian Test Symposium*, November 2009.

[C14] M. Hunger, S. Hellebrand, A. Czutro, I. Polian and B. Becker, "ATPG-Based Grading of Strong Fault-Secureness," in *IEEE International On-line Testing Symposium*, June 2009.

[C15] N. Houarche, A. Czutro, M. Comte, P. Engelke, I. Polian, B. Becker and M. Renovell, "An Electrical Model for the Fault Simulation of Small-Delay Faults Caused by Crosstalk Aggravated Resistive Short Defects," in *VLSI Test Symposium*, April 2009.

[C16] A. Czutro, I. Polian, M. Lewis, P. Engelke, S.M. Reddy and B. Becker, "TIGUAN: Thread-parallel Integrated test pattern Generator Utilizing satisfiability ANalysis," in *International Conference on VLSI Design*, pp. 227–232, January 2009.

[C17] A. Czutro, N. Houarche, P. Engelke, I. Polian, M. Comte, M. Renovell and B. Becker, "A Simulator of Small-Delay Faults Caused by Resistive-Open Defects," in *IEEE European Test Symposium*, pp. 113–118, May 2008.

[C18] I. Polian, A. Czutro, S. Kundu and B. Becker, "Power Droop Testing," in *International Conference on Computer Design*, pp. 243–250, October 2006.

[C19] I. Polian, A. Czutro and B. Becker, "Evolutionary Optimization in Code-Based Test Compression," in *Conference on Design, Automation and Test in Europe*, pp. 1124–1129, March 2005.

Workshop contributions (refereed)

[W1] L. Feiten, M. Sauer, T. Schubert, A. Czutro, V. Tomashevich, E. Böhl, I. Polian and B. Becker, "#SAT for Vulnerability Analysis of Security Components," in *IEEE European Test Symposium (informal proceedings)*, May 2013.

[W2] A. Czutro, M. Sauer, I. Polian and B. Becker, "Multi-Conditional ATPG using SAT with Preferences," in *GI/ITG Workshop "Testmethoden und Zuverlässigkeit von Schaltungen und Systemen"*, February 2012.

[W3] M. Sauer, S. Kupferschmid, A. Czutro, I. Polian, S.M. Reddy and B. Becker, "Functional Justification in Sequential Circuits using SAT and Craig Interpolation," in *GI/ITG Workshop "Testmethoden und Zuverlässigkeit von Schaltungen und Systemen"*, February 2012.

[W4] J. Jiang, M. Sauer, A. Czutro, B. Becker and I. Polian, "On the Optimality of K Longest Path Generation," in *Workshop on RTL and High Level Testing*, November 2011.

[W5] A. Czutro, B. Becker and I. Polian, "Performance Evaluation of SAT-Based ATPG on Multi-Core Architectures," in *IEEE East-West Design & Test Symposium*, September 2009.

[W6] M. Hunger, S. Hellebrand, A. Czutro, I. Polian and B. Becker, "Robustheitsanalyse stark fehlersicherer Schaltungen mit SAT-basierter Testmustererzeugung," in *GMM/ITG-Fachtagung "Zuverlässigkeit und Entwurf"*, September 2009.

[W7] A. Czutro, B. Becker and I. Polian, "Performance Evaluation of SAT-Based Automatic Test Pattern Generation on Multi-Core Architectures," in *GI/ITG International Conference on Architecture of Computing Systems, Many-Cores Workshop*, March 2009.

[W8] N. Houarche, A. Czutro, M. Comte, P. Engelke, I. Polian, B. Becker and M. Renovell, "Deriving an Electrical Model for Delay Faults Caused by Crosstalk Aggravated Resistive Short Defects," in *Latin-American Test Workshop*, March 2009.

[W9] A. Czutro, B. Becker and I. Polian, "Performance Evaluation of SAT-Based ATPG on Multi-Core Architectures," in *GI/ITG Workshop "Testmethoden und Zuverlässigkeit von Schaltungen und Systemen"*, February 2009. Poster.

[W10] A. Czutro, I. Polian, M. Lewis, P. Engelke, S.M. Reddy and B. Becker, "TIGUAN: Thread-parallel Integrated test pattern Generator Utilizing satisfiability ANalysis," in *edaWorkshop*, May 2008. Poster.

[W11] I. Polian, B. Becker and A. Czutro, "Compression Methods for Path Delay Fault Test Pair Sets: A Comparative Study," in *IEEE European Test Symposium*, pp. 263–264, May 2004. Poster.

Bibliography

[1] http://minisat.se

[2] http://tiguan-tsi.czutro.de

[3] http://www.cs.ubc.ca/labs/beta/projects/paramils

[4] http://www.intel.com/pressroom/kits/quickrefyr.htm

[5] http://www.nxp.com

[6] http://www.samsung.com/global/business/semiconductor

[7] "Panel 'ITC'99 Benchmark Circuits — Preliminary Results'," in *International Test Conference*, pp. 1125–1130, 1999.

[8] "Special Issue on 'Process Variation and Stochastic Design and Test'," *IEEE Design & Test of Computers*, vol. 23, pp. 434–520, June 2006.

[9] *Test and Test Equipment.* International Technology Roadmap For Semiconductors, 2009.

[10] *Test and Test Equipment.* International Technology Roadmap For Semiconductors, 2011.

[11] M. Abramovici, "DOs and DON'Ts in Computing Fault Coverage," in *International Test Conference*, 1993.

[12] M. Abramovici, M.A. Breuer and A.D. Friedman, *Digital Systems Testing and Testable Design.* Computer Science Press, 1990.

[13] M. Abramovici, J.J. Kulikowski, P.R. Menon and D.T. Miller, "SMART And FAST: Test Generation for VLSI Scan-Design Circuits," *IEEE Design & Test of Computers*, vol. 3, pp. 43–54, August 1986.

[14] M. Abramovici, P.R. Menon and D.T. Miller, "Critical Path Tracing — An Alternative to Fault Simulation," in *IEEE Design Automation Conference*, pp. 214–220, June 1983.

[15] J.M. Acken and S.D. Millman, “Fault Model Evolution for Diagnosis; Accuracy vs Precision,” in *Custom Integrated Circuits Conference*, pp. 13.4.1–13.4.4, 1992.

[16] R.C. Aitken, “New Defect Behavior at 130nm and Beyond,” in *IEEE European Test Symposium*, pp. 279–284, 2004.

[17] S.B. Akers, C. Joseph and B. Krishnamurthy, “On the Role of Independent Fault Sets in the Generation of Minimal Test Sets,” in *International Test Conference*, pp. 204–209, 1987.

[18] D.B. Armstrong, “A Deductive Method for Simulating Faults in Logic Circuits,” *IEEE Transactions on Computers*, vol. C-21, pp. 464–471, May 1972.

[19] G. Audemard and L. Simon, “Predicting Learnt Clauses Quality in Modern SAT Solvers,” in *International Joint Conference on Artificial Intelligence*, pp. 399–404, 2009.

[20] B. Ayari and B. Kaminska, “A New Dynamic Test Vector Compaction for Automatic Test Pattern Generation,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 3, pp. 353–358, 1994.

[21] O. Bailleux and P. Marquis, “DISTANCE-SAT: Complexity and Algorithms,” in *AAAI National Conference on Artificial Intelligence*, 1999.

[22] K. Baker and M. Nourani, “Interconnect Test Pattern Generation Algorithm for Meeting Device and Global SSO Limits With Safe Initial Vectors,” in *International Test Conference*, 2004.

[23] P. Banerjee and J.A. Abraham, “A Multivalued Algebra for Modeling Physical Failures in MOS VLSI Circuits,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 4, no. 5, pp. 312–321, 1985.

[24] P. Barth, “A Davis-Putnam Based Enumeration Algorithm for Linear Pseudo-Boolean Optimization,” Technical Report MPI-I-95-2-003, Max-Planck-Institute for Computer Science, Saarbrücken, 1995.

[25] R.C. Baumann, “Soft Errors in Advanced Computer Systems,” *IEEE Design & Test of Computers*, vol. 22, no. 3, pp. 258–266, 2005.

[26] R.J. Bayardo and R.C. Schrag, “Using CSP Look-Back Techniques to Solve Real-World SAT Instances,” in *AAAI National Conference on Artificial Intelligence*, 1997.

[27] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita and Y. Zhu, "Symbolic Model Checking Using SAT Procedures Instead of BDDs," in *IEEE Design Automation Conference*, 1999.

[28] S. Borkar, "Designing Reliable Systems from Unreliable Components: the Challenges of Transistor Variability and Degradation," *IEEE Microelectronics Journal*, vol. 25, no. 6, pp. 10–16, 2005.

[29] D. Brand, "Verification of Large Synthesized Designs," in *International Conference on Computer-Aided Design*, pp. 534–537, 1993.

[30] M.A. Breuer, S.K. Gupta and S. Nazarian, "Efficient Identification of Crosstalk Induced Slowdown Targets," in *IEEE Asian Test Symposium*, pp. 124–131, November 2004.

[31] F. Brglez, D. Bryan and K. Koźmiński, "Combinational Profiles of Sequential Benchmark Circuits," in *IEEE International Symposium on Circuits and Systems*, pp. 1929–1934, 1989.

[32] F. Brglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Circuits and a Target Translator in Fortran," in *IEEE International Symposium on Circuits and Systems*, pp. 663–698, 1985.

[33] R.E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 8, pp. 677–691, 1986.

[34] M.L. Bushnell and V.D. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits.* Kluwer Academic Publishers, 2001.

[35] N. Butt, K. McStay, A. Cestero, H. Ho, W. Kong, S. Fang, R. Krishnan, B. Khan, A. Tessier, W. Davies, S. Lee, Y. Zhang, J. Johnson, S. Rombawa, R. Takalkar, A. Blauberg, K.V. Hawkins, J. Liu, S. Rosenblatt, P. Goyal, S. Gupta, J. Ervin, Z. Li, S. Galis, J. Barth, M. Yin, T. Weaver, J.H. Li, S. Narasimha, P. Parries, W.K. Henson, N. Robson, T. Kirihata, M. Chudzik, E. Maciejewski, P. Agnello, S. Stiffler and S.S. Iyer, "A 0.039um2 High Performance eDRAM Cell Based on 32nm High-K/Metal SOI Technology," in *IEEE International Electron Devices Meeting*, pp. 27.5.1–27.5.4, 2010.

[36] T. Bäck, *Evolutionary Algorithms in Theory and Practice.* Oxford University Press, 1996.

[37] J.L. Carter, V.S. Iyengar and B.K. Rosen, "Efficient Test Coverage Determination for Delay Faults," in *International Test Conference*, pp. 418–427, 1987.

[38] J.T.Y. Chang, C.-W. Tseng, C.M.J. Li, M. Purtell and E.J. McCluskey, “Analysis of Pattern-Dependent and Timing-Dependent Failures in an Experimental Test Chip,” in *International Test Conference*, pp. 184–193, 1998.

[39] Y.-S. Chang, S.K. Gupta and M.A. Breuer, “Analysis of Ground Bounce in Deep Sub-Micron Circuits,” in *VLSI Test Symposium*, pp. 110–116, 1997.

[40] Y.-S. Chang, S.K. Gupta and M.A. Breuer, “Test Generation for Ground Bounce in Internal Logic Circuitry,” in *VLSI Test Symposium*, pp. 95–104, 1999.

[41] Y.-S. Chang, S.K. Gupta and M.A. Breuer, “Test Generation for Maximizing Ground Bounce for Internal Circuitry with Reconvergent Fan-outs,” in *VLSI Test Symposium*, pp. 358–366, 2001.

[42] W.Y. Chen, S.K. Gupta and M.A. Breuer, “Test Generation for Crosstalk-Induced Faults: Framework and Computational Results,” *Journal of Electronic Testing*, vol. 18, pp. 17–28, January 2002.

[43] K.Y. Cho, S. Mitra and E.J. McCluskey, “Gate Exhaustive Testing,” in *International Test Conference*, 2005.

[44] S.H. Choi, B.C. Paul and K. Roy, “Novel Sizing Algorithm for Yield Improvement under Process Variation in Nanometer Technology,” in *IEEE Design Automation Conference*, pp. 454–459, 2004.

[45] J. Chung, J. Xiong, V. Zolotov and J.A. Abraham, “Testability Driven Statistical Path Selection,” in *IEEE Design Automation Conference*, pp. 417–422, June 2011.

[46] E.M. Clarke, A. Biere, R. Raimi and Y. Zhu, “Bounded Model Checking Using Satisfiability Solving,” *Formal Methods in System Design*, vol. 19, no. 1, pp. 7–34, 2001.

[47] S.A. Cook, “The Complexity of Theorem-Proving Procedures,” in *ACM Symposium on Theory of Computing*, pp. 151–158, 1971.

[48] F. Corno, M. Sonza Reorda and G. Squillero, “RT-Level ITC 99 Benchmarks and First ATPG Results,” *IEEE Design & Test of Computers*, vol. 17, pp. 44–53, July 2000.

[49] B. Courtois, “Failure Mechanisms, Fault Hypotheses and Analytical Testing of LSI-NMOS (HMOS) Circuits,” in *International Conference on VLSI*, pp. 341–350, 1981.

[50] N. Creignou, S. Khanna and M. Sudan, *Complexity Classifications of Boolean Constraint Satisfaction Problems.* Society for Industrial and Applied Mathematics, 2001.

[51] A. Czutro, N. Houarche, P. Engelke, I. Polian, M. Comte, M. Renovell and B. Becker, "A Simulator of Small-Delay Faults Caused by Resistive-Open Defects," in *IEEE European Test Symposium*, pp. 113–118, May 2008.

[52] A. Czutro, M. Imhof, J. Jiang, A. Mumtaz, M. Sauer, B. Becker, I. Polian and H.-J. Wunderlich, "Variation-Aware Fault Grading," in *IEEE Asian Test Symposium*, pp. 344–349, November 2012.

[53] A. Czutro, I. Polian, P. Engelke, S.M. Reddy and B. Becker, "Dynamic Compaction in SAT-Based ATPG," in *IEEE Asian Test Symposium*, November 2009.

[54] A. Czutro, I. Polian, M. Lewis, P. Engelke, S.M. Reddy and B. Becker, "TIGUAN: Thread-parallel Integrated test pattern Generator Utilizing satisfiability ANalysis," in *International Conference on VLSI Design*, pp. 227–232, January 2009.

[55] A. Czutro, I. Polian, M. Lewis, P. Engelke, S.M. Reddy and B. Becker, "Thread-Parallel Integrated Test Pattern Generator Utilizing Satisfiability Analysis," *International Journal of Parallel Programming*, vol. 38, pp. 185–202, June 2010.

[56] A. Czutro, M. Sauer, I. Polian and B. Becker, "Multi-Conditional SAT-ATPG for Power-Droop Testing," in *IEEE European Test Symposium*, May 2012.

[57] A. Czutro, M. Sauer, T. Schubert, I. Polian and B. Becker, "SAT-ATPG Using Preferences for Improved Detection of Complex Defect Mechanisms," in *VLSI Test Symposium*, April 2012.

[58] A. Datta, S. Bhunia, J.H. Choi, S. Mukhopadhyay and K. Roy, "Speed Binning Aware Design Methodology to Improve Profit under Parameter Variations," in *ASP Design Automation Conference*, 2006.

[59] M. Davis, G. Logemann and D. Loveland, "A Machine Program for Theorem Proving," *Communications of the ACM*, vol. 5, pp. 394–397, 1962.

[60] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *Journal of the ACM*, vol. 7, no. 3, pp. 201–215, 1960.

[61] A. Davoodi and A. Srivastava, "Probabilistic Dual-Vth Leakage Optimization Under Variability," in *International Symposium on Low Power Electronics and Design*, pp. 143–148, 2005.

[62] A. Davoodi and A. Srivastava, “Variability Driven Gate Sizing for Binning Yield Optimization,” in *IEEE Design Automation Conference*, pp. 959–964, 2006.

[63] B.I. Dervisoglu and G.E. Stong, “Design for Testability: Using Scanpath Techniques for Path-Delay Test and Measurement,” in *International Test Conference*, pp. 365–374, 1991.

[64] R. Desineni, K.N. Dwarkanath and R.D. Blanton, “Universal Test Generation Using Fault Tuples,” in *International Test Conference*, pp. 812–819, 2000.

[65] E. Di Rosa, E. Giunchiglia and M. Maratea, “A New Approach for Solving Satisfiability Problems with Qualitative Preferences,” in *European Conference on Artificial Intelligence*, pp. 510–514, 2008.

[66] R. Drechsler, *Evolutionary Algorithms for VLSI CAD.* Kluwer Academic Publisher, 1998.

[67] R. Drechsler, S. Eggersglüß, G. Fey, A. Glowatz, F. Hapke, J. Schlöffel and D. Tille, “On Acceleration of SAT-Based ATPG for Industrial Designs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1329–1333, 2008.

[68] S. Eggersglüß and R. Drechsler, “Improving Test Pattern Compactness in SAT-Based ATPG,” in *IEEE Asian Test Symposium*, pp. 445–452, 2007.

[69] S. Eggersglüß and R. Drechsler, “Robust Algorithms for High Quality Test Pattern Generation Using Boolean Satisfiability,” in *International Test Conference*, pp. 1–10, November 2010.

[70] S. Eggersglüß, D. Tille and R. Drechsler, “Speeding Up SAT-Based ATPG Using Dynamic Clause Activation,” in *IEEE Asian Test Symposium*, pp. 177–182, November 2009.

[71] A.H. El-Maleh and K. Al-Utaibi, “An Efficient Test Relaxation Technique for Synchronous Sequential Circuits,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 6, pp. 933–940, 2004.

[72] R.D. Eldred, “Test Routines Based on Symbolic Logical Statements,” *Journal of the ACM*, vol. 6, no. 1, pp. 33–36, 1959.

[73] P. Engelke, B. Becker, M. Renovell, J. Schlöffel, B. Braitling and I. Polian, “SUPERB: Simulator Utilizing Parallel Evaluation of Resistive Bridges,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 14, pp. 56:1–56:21, August 2009.

[74] P. Engelke, B. Braitling, I. Polian, M. Renovell and B. Becker, "SUPERB: Simulator Utilizing Parallel Evaluation of Resistive Bridges," in *IEEE Asian Test Symposium*, pp. 433–438, 2007.

[75] P. Engelke, I. Polian, M. Renovell and B. Becker, "Simulating Resistive Bridging and Stuck-At Faults," in *International Test Conference*, pp. 1051–1059, 2003.

[76] P. Engelke, I. Polian, M. Renovell and B. Becker, "Automatic test pattern generation for resistive bridging faults," in *IEEE European Test Symposium*, pp. 160–165, 2004.

[77] P. Engelke, I. Polian, M. Renovell and B. Becker, "Automatic Test Pattern Generation for Resistive Bridging Faults," *Journal of Electronic Testing*, vol. 22, pp. 61–69, February 2006.

[78] P. Engelke, I. Polian, M. Renovell and B. Becker, "Simulating Resistive Bridging and Stuck-At Faults," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, pp. 2181–2192, October 2006.

[79] N. Eén and A. Biere, "Effective Preprocessing in SAT Through Variable and Clause Elimination," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 61–75, Springer, 2005.

[80] N. Eén and N. Sörensson, "An Extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, vol. 2919 of *Lecture Notes in Computer Science*, pp. 541–638, Springer, 2003.

[81] N. Eén and N. Sörensson, "Temporal Induction by Incremental SAT Solving," in *International Workshop on Bounded Model Checking*, vol. 89, pp. 541–638, Elsevier, 2003.

[82] L. Feiten, M. Sauer, T. Schubert, A. Czutro, E. Böhl, I. Polian and B. Becker, "#SAT-Based Vulnerability Analysis of Security Components — A Case Study," in *International Symposium on Defect and Fault Tolerance*, October 2012.

[83] L. Feiten, M. Sauer, T. Schubert, A. Czutro, V. Tomashevich, E. Böhl, I. Polian and B. Becker, "#SAT for Vulnerability Analysis of Security Components," in *IEEE European Test Symposium (informal proceedings)*, May 2013.

[84] F.J. Ferguson and T. Larrabee, "Test Pattern Generation for Realistic Bridge Fault in CMOS ICs," in *International Test Conference*, pp. 492–499, 1991.

[85] F.J. Ferguson and J. Shen, "Extraction and Simulation of Realistic CMOS Faults Using Inductive Fault Analysis," in *International Test Conference*, pp. 475–484, 1988.

[86] G. Fey, T. Warode and R. Drechsler, "Reusing Learned Information in SAT-Based ATPG," in *International Conference on VLSI Design*, pp. 69–76, 2007.

[87] H. Fujiwara, "FAN: A Fanout-Oriented Test Pattern Generation Algorithm," in *IEEE International Symposium on Circuits and Systems*, pp. 671–674, 1985.

[88] H. Fujiwara and T. Inoue, "Optimal Granularity of Test Generation in a Distributed System," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, pp. 885–892, August 1990.

[89] H. Fujiwara and T. Shimono, "On the Acceleration of Test Generation Algorithms," *IEEE Transactions on Computers*, vol. 32, pp. 1137–1144, December 1983.

[90] J.M. Galey, R.E. Norby and J.P. Roth, "Techniques for the Diagnosing of Switching Circuit Failures," in *Symposium on Switching Circuit Theory and Logical Design*, pp. 152–160, March 1961.

[91] J. Galiay, Y. Crouzet and M. Vergniault, "Physical Versus Logical Fault Models MOS LSI Circuits: Impact on Their Testability," *IEEE Transactions on Computers*, vol. c-29, pp. 527–531, June 1980.

[92] K. Ganeshpure and S. Kundu, "On ATPG for Multiple Aggressor Crosstalk Faults," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, pp. 774–787, May 2010.

[93] J. Giraldi and M.L. Bushnell, "EST: The New Frontier in Automatic Test-Pattern Generation," in *International Conference on Computer-Aided Design*, pp. 667–672, 1991.

[94] P. Girard, C. Landrault, S. Pravossoudovitch and B. Rodríguez, "A Diagnostic ATPG for Delay Faults Based on Genetic Algorithms," in *International Test Conference*, pp. 286–293, 1996.

[95] E. Giunchiglia and M. Maratea, "Solving Optimization Problems with DLL," in *European Conference on Artificial Intelligence*, pp. 377–381, 2006.

[96] E. Giunchiglia and M. Maratea, "Planning as Satisfiability with Preferences," in *AAAI National Conference on Artificial Intelligence*, pp. 987–992, 2007.

[97] E. Gizdarski and H. Fujiwara, "SPIRIT: A Highly Robust Combinational test Generation Algorithm," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, pp. 1446–1458, December 2002.

[98] P. Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, pp. 215–222, March 1981.

[99] P. Goel and B.C. Rosales, "Test Generation and Dynamic Compaction of Tests," in *International Test Conference*, pp. 189–192, 1979.

[100] P. Goel and B.C. Rosales, "Dynamic Test Compaction with Fault Selection Using Sensitizable Path Tracing," *IBM Technical Disclosure Bulletin*, vol. 23, pp. 1954–1957, October 1980.

[101] P. Goel and B.C. Rosales, "PODEM-X: An Automatic Test Generation System for VLSI Logic Structures," in *IEEE Conference on Design Automation*, pp. 260–268, June 1981.

[102] D.E. Goldberg, *Genetic Algorithms in Search, Optimization & Machine Learning.* Addision-Wesley Publisher Company, 1989.

[103] E. Goldberg and Y. Novikov, "BerkMin: A Fast and Robust SAT-Solver," in *Conference on Design, Automation and Test in Europe*, pp. 142–149, March 2002.

[104] S.S. Gorshe and B. Bose, "A Self-Checking ALU Design with Efficient Codes," in *VLSI Test Symposium*, pp. 157–161, 1996.

[105] B. Greene, Q. Liang, K. Amarnath, Y. Wang, J. Schaeffer, M. Cai, Y. Liang, S. Saroop, J. Cheng, A. Rotondaro, S.-J. Han, R. Mo, K. McStay, S. Ku, R. Pal, M. Kumar, B. Dirahoui, B. Yang, F. Tamweber, W.-H. Lee, M. Steigerwalt, H. Weijtmans, J. Holt, L. Black, S. Samavedam, M. Turner, K. Ramani, D. Lee, M. Belyansky, M. Chowdhury, D. Aime, B. Min, H. van Meer, H. Yin, K. Chan, M. Angyal, M. Zaleski, O. Ogunsola, C. Child, L. Zhuang, H. Yan, D. Permanaa, J. Sleight, D. Guo, S. Mittl, D. Ioannou, E. Wu, M. Chudzik, D.-G. Park, D. Brown, S. Luning, D. Mocuta, E. Maciejewski, K. Henson and E. Leobandung, "High Performance 32nm SOI CMOS with High-k/Metal Gate and 0.149 µm2 SRAM and Ultra Low-k Back End with Eleven Levels of Copper," in *Symposium on VLSI Technology*, pp. 140–141, 2009.

[106] P. Gupta and A.B. Kahng, "Manufacturing-Aware Physical Design," in *IEEE International Conference on Computer-Aided Design*, pp. 681–687, 2003.

[107] R. Gómez, A. Girón and V.H. Champac, "Test of Interconnection Opens Considering Coupling Signals," in *International Symposium on Defect and Fault Tolerance*, pp. 247–255, 2005.

[108] I. Hamzaoglu and J.H. Patel, "New Techniques for Deterministic Test Pattern Generation," *Journal of Electronic Testing*, vol. 15, pp. 63–73, 1999.

[109] C.F. Hawkins, J. Soden, A. Righter and F.J. Ferguson, "Defect Classes — An Overdue Paradigm for CMOS IC Testing," in *International Test Conference*, pp. 413–425, 1994.

[110] J.P. Hayes, "Fault Modeling for Digital MOS Integrated Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 3, no. 3, pp. 200–208, 1984.

[111] J.P. Hayes, *Computer Architecture and Organization*. McGraw-Hill, 1998.

[112] Z. He, T. Lv, H. Li and X. Li, "An Efficient Algorithm for Finding a Universal Set of Testable Long Paths," in *IEEE Asian Test Symposium*, pp. 319–324, 2010.

[113] K. Heragu, J.H. Patel and V.D. Agrawal, "Segment Delay Faults: A New Fault Model," in *VLSI Test Symposium*, pp. 32–39, 1996.

[114] M. Herbstritt, B. Becker and C. Scholl, "Advanced SAT-Techniques for Bounded Model Checking of Blackbox Designs," in *International Workshop on Microprocessor Test and Verification*, pp. 37–44, IEEE Computer Society, 2006.

[115] M.J.H. Heule, M. Dufour, J. van Zwieten and H. van Maaren, "March_eq: Implementing Additional Reasoning into an Efficient Look-Ahead SAT Solver," in *Theory and Applications of Satisfiability Testing*, 2005.

[116] S. Hillebrecht, *Interconnect Reliability and Test*. Design, Test and Verification of Embedded Systems, Der Andere Verlag, 2010.

[117] S. Hillebrecht, M. Kochte, H.-J. Wunderlich and B. Becker, "Exact Stuck-at Fault Classification in Presence of Unknowns," in *IEEE European Test Symposium*, May 2012.

[118] S. Hillebrecht, I. Polian, P. Engelke, B. Becker, M. Keim and W.-T. Cheng, "Extraction, Simulation and Test Generation for Interconnect Open Defects Based on Enhanced Aggressor-Victim Model," in *International Test Conference*, pp. 1–10, 2008.

[119] S. Holst and H.-J. Wunderlich, "Adaptive Debug and Diagnosis without Fault Dictionaries," in *IEEE European Test Symposium*, pp. 7–12, May 2007.

[120] J.N. Hooker, "Solving the Incremental Satisfiability Problem," *Journal of Logic Programming*, vol. 15, no. 1-2, pp. 177–186, 1993.

[121] J.N. Hooker and H. Yan, "Logic Circuit Verification by Benders Decomposition," technical report, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh PA, 1991.

[122] M.J. Howes and D.V. Morgan, *Reliability and Degradation — Semiconductor Devices and Circuits*. Wiley-Interscience, 1981.

[123] M.S. Hsiao, E.M. Rudnick and J.H. Patel, "Effects of Delay Models on Peak Power Estimation of VLSI Sequential Circuits," in *International Conference on Computer-Aided Design*, pp. 45–51, November 1997.

[124] L.M. Huisman, "Diagnosing Arbitrary Defects in Logic Designs Using Single Location at a Time (SLAT)," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, pp. 91–101, January 2004.

[125] M. Hunger and S. Hellebrand, "Verification and Analysis of Self-Checking Properties through ATPG," in *IEEE International On-line Testing Symposium*, pp. 25–30, 2008.

[126] M. Hunger, S. Hellebrand, A. Czutro, I. Polian and B. Becker, "ATPG-Based Grading of Strong Fault-Secureness," in *IEEE International On-line Testing Symposium*, June 2009.

[127] F. Hutter, H.H. Hoos, K. Leyton-Brown and T. Stützle, "ParamILS: An Automatic Algorithm Configuration Framework," *Journal of Artificial Intelligence Research*, vol. 36, pp. 267–306, October 2009.

[128] O.H. Ibarra and S.K. Sahni, "Polynomially Complete Fault Detection Problems," *IEEE Transactions on Computers*, vol. 24, pp. 242–249, March 1975.

[129] N.K. Jha and S.K. Gupta, *Testing of Digital Systems*. Cambridge University Press, 2003.

[130] J. Jiang, M. Sauer, A. Czutro, B. Becker and I. Polian, "On the Optimality of K Longest Path Generation," in *Workshop on RTL and High Level Testing*, November 2011.

[131] J. Jiang, M. Sauer, A. Czutro, B. Becker and I. Polian, "On the Optimality of K Longest Path Generation Algorithm Under Memory Constraints," in *Conference on Design, Automation and Test in Europe*, pp. 418–423, March 2012.

[132] Y.-M. Jiang and K.-T. Cheng, "Analysis of Performance Impact Caused by Power Supply Noise in Deep Submicron Devices," in *IEEE Design Automation Conference*, pp. 760–765, 1999.

[133] S. Kajihara and K. Miyase, "On Identifying Don't Care Inputs of Test Patterns for Combinational Circuits," in *International Conference on Computer-Aided Design*, pp. 364–369, 2001.

[134] S. Kajihara, I. Pomeranz, K. Kinoshita and S.M. Reddy, "Cost-Effective Generation of Minimal Test Sets for Stuck-at Faults in Combinational Logic Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 12, pp. 1496–1504, 1995.

[135] R.M. Karp, "Reducibility Among Combinatorial Problems," in *Complexity of Computer Computations*, pp. 85–103, Plenum Press, 1972.

[136] H.A. Kautz and B. Selman, "Planning as Satisfiability," in *European Conference on Artificial Intelligence*, pp. 359–363, 1992.

[137] M. Keim, N. Drechsler, R. Drechsler and B. Becker, "Combining GAs and Symbolic Methods for High Quality Tests of Sequential Circuits," *Journal of Electronic Testing*, vol. 17, no. 1, pp. 37–51, 2001.

[138] M. Ketkar and E. Chiprout, "A Microarchitecture-Based Framework for Pre- and Post-silicon Power Delivery Analysis," in *IEEE International Symposium on Microarchitecture*, pp. 179–188, December 2009.

[139] C.H. Kim, K. Roy, S. Hsu, A. Alvandpour, R.K. Krishnamurthy and S. Borkar, "A Process Variation Compensating Technique for Sub-90 nm Dynamic Circuits," in *Symposium on VLSI Circuits*, pp. 205–206, June 2003.

[140] T. Kirkland and M.R. Mercer, "A Topological Search Algorithm for ATPG," in *IEEE Design Automation Conference*, pp. 502–508, 1987.

[141] I. Koren and M. Krishna, *Fault-Tolerant Systems*. Morgan Kaufmann, 2007.

[142] A. Krstić, Y.-M. Jiang and K.-T. Cheng, "Pattern Generation for Delay Testing and Dynamic Timing Analysis Considering Power-Supply Noise Effects," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 3, pp. 416–425, 2001.

[143] R. Kundu and R.D. Blanton, "Identification of Crosstalk Switch Failures in Domino CMOS Circuits," in *International Test Conference*, pp. 502–509, 2000.

[144] S. Kundu, S.T. Zachariah, Y.-S. Chang and C. Tirumurti, "On Modeling Crosstalk Faults," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 12, pp. 1909–1915, 2005.

[145] B. Könemann, “LFSR-Coded Test Patterns for Scan Designs,” in *European Design and Test Conference*, pp. 237–242, 1992.

[146] M. Ladjadj and J.F. McDonald, “Benchmark Runs of the Subscripted D-Algorithm with Observation Path Mergers on the Brglez-Fujiwara Circuits,” in *IEEE Conference on Design Automation*, pp. 509–515, June 1987.

[147] T. Larrabee, “Efficient Generation of Test Patterns Using Boolean Difference,” in *International Test Conference*, pp. 795–801, 1989.

[148] T. Larrabee, “Test Pattern Generation Using Boolean Satisfiability,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, pp. 4–15, 1992.

[149] K.T. Lee, C. Nordquist and J.A. Abraham, “Automatic Test Pattern Generation for Crosstalk Glitches in Digital Circuits,” in *VLSI Test Symposium*, pp. 34–39, 1998.

[150] W.-P. Lee, H.-Y. Liu and Y.-W. Chang, “Voltage Island Aware Floorplanning for Power and Timing Optimization,” in *International Conference on Computer-Aided Design*, pp. 389–394, 2006.

[151] M. Lewis, *SAT, QBF and Multicore Processors.* Design, Test and Verification of Embedded Systems, Der Andere Verlag, 2011.

[152] M. Lewis, T. Schubert and B. Becker, “Multithreaded SAT Solving,” in *ASP Design Automation Conference*, pp. 926–921, January 2007.

[153] M. Lewis, T. Schubert and B. Becker, “DPLL-based Reasoning in a Multi-Core Environment,” in *International Workshop on Microprocessor Test and Verification*, 2009.

[154] X. Liang, G.-Y. Wei and D. Brooks, “ReVIVaL: A Variation-Tolerant Architecture Using Voltage Interpolation and Variable Latency,” in *International Symposium on Comp. Architecture*, pp. 191–202, June 2008.

[155] C.J. Lin and S.M. Reddy, “On Delay Fault Testing in Logic Circuits,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, no. 5, pp. 694–703, 1987.

[156] J.-C. Lo, S. Thanawastien, T.R.N. Rao and M. Nicolaidis, “An SFS Berger Check Prediction ALU and its Application to Self-Checking Processor Designs,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 4, pp. 525–540, 1992.

[157] M. Luby, A. Sinclair and D. Zuckerman, "Optimal Speedup of Las Vegas Algorithms," *Information Processing Letters*, vol. 47, pp. 173–180, 1993.

[158] D. Macemon, "User Defined Fault Models," September 2011. White Paper Mentor Graphics — http://www.mentor.com/products/silicon-yield/techpubs/user-defined-fault-models-70606.

[159] U. Mahlstedt, T. Grüning, C. Özcan and W. Daehn, "CONTEST: A Fast ATPG Tool for Very Large Combinational Circuits," in *International Conference on Computer-Aided Design*, pp. 222–225, 1990.

[160] A.K. Majhi and V.D. Agrawal, "Tutorial: Delay Fault Models and Coverage," in *International Conference on VLSI Design*, pp. 364–369, 1998.

[161] W. Maly, "Realistic Fault Modeling for VLSI Testing," in *IEEE Design Automation Conference*, pp. 173–180, 1987.

[162] J.P. Marques-Silva, "The Impact of Branching Heuristics in Propositional Satisfiability Algorithms," in *Portuguese Conference on Artificial Intelligence*, 1999.

[163] J.P. Marques-Silva and K.A. Sakallah, "GRASP — A New Search Algorithm for Satisfiability," in *International Conference on Computer-Aided Design*, pp. 220–227, 1996.

[164] J.P. Marques-Silva and K.A. Sakallah, "Robust Search Algorithms for Test Pattern Generation," in *International Symposium on Fault-Tolerant Computing*, pp. 152–161, June 1997.

[165] J.P. Marques-Silva and K.A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 506–521, 1999.

[166] J.P. Marques-Silva and K.A. Sakallah, "Boolean Satisfiability in Electronic Design Automation," in *IEEE Design Automation Conference*, pp. 675–680, 2000.

[167] P.C. Maxwell and R.C. Aitken, "Biased Voting: A Method for Simulating CMOS Bridging Faults in the Presence of Variable Gate Logic Thresholds," in *International Test Conference*, pp. 63–72, 1993.

[168] D. McAllester, B. Selman and H.A. Kautz, "Evidence for Invariants in Local Search," in *AAAI National Conference on Artificial Intelligence*, pp. 321–326, 1997.

[169] E.J. McCluskey, "Quality and Single-Stuck Faults," in *International Test Conference*, p. 597, 1993.

[170] J.F. McDonald and C. Benmehrez, "Test Set Reduction Using the Subscripted D-Algorithm," in *International Test Conference*, pp. 115–121, August 1983.

[171] G.H. Mealy, "A Method to Synthesizing Sequential Circuits," *Bell Systems Technical Journal*, pp. 1045–1079, 1955.

[172] P.R. Menon, H. Ahuja and M. Harihara, "Redundancy Identification and Removal in Combinational Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, pp. 646–651, May 1994.

[173] P.R. Menon and M. Harihara, "Identification of Undetectable Faults in Combinational Circuits," in *International Conference on Computer Design*, pp. 290–293, October 1989.

[174] S. Mitra, N. Seifert, M. Zhang, Q. Shi and K.S. Kim, "Robust System Design with Built-In Soft-Error Resilience," *IEEE Computer*, vol. 38, no. 2, pp. 43–52, 2005.

[175] K. Mohanram and N.A. Touba, "Cost-Effective Approach for Reducing Soft Error Failure Rate in Logic Circuits," in *International Test Conference*, pp. 893–901, 2003.

[176] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *IEEE Design Automation Conference*, 2001.

[177] A. Muhtaroglu, G. Taylor and T. Rahal-Arabi, "On-Die Droop Detector for Analog Sensing of Power Supply Noise," *Journal of Solid-State Circuits*, vol. 39, no. 4, pp. 651–660, 2004.

[178] K. Pipatsrisawat and A. Darwiche, "RSAT 3.01 — Solver Description," in *SAT Race*, 2008.

[179] I. Polian, *On Non-standard Fault Models for Logic Digital Circuits: Simulation, Design for Testability, Industrial Applications*. VDI-Verlag, March 2004.

[180] I. Polian, B. Becker and S.M. Reddy, "Evolutionary Optimization of Markov Sources for Pseudo Random Scan BIST," in *Conference on Design, Automation and Test in Europe*, pp. 1184–1185, 2003. Poster.

[181] I. Polian, A. Czutro and B. Becker, "Evolutionary Optimization in Code-Based Test Compression," in *Conference on Design, Automation and Test in Europe*, pp. 1124–1129, March 2005.

[182] I. Polian, A. Czutro, S. Kundu and B. Becker, "Power Droop Testing," *IEEE Design & Test of Computers*, vol. 24, pp. 276–284, May 2007.

[183] I. Polian, P. Engelke and B. Becker, "Efficient Bridging Fault Simulation of Sequential Circuits Based on Multi-Valued Logics," in *International Symposium on Multi-Valued Logic*, pp. 216–222, 2002.

[184] I. Polian, P. Engelke, M. Renovell and B. Becker, "Modelling Feedback Bridging Faults With Non-Zero Resistance," in *European Test Workshop*, pp. 91–96, 2003.

[185] I. Pomeranz, L.N. Reddy and S.M. Reddy, "COMPACTEST: A Method to Generate Compact Test Sets for Combinational Circuits," in *International Test Conference*, pp. 194–203, 1991.

[186] I. Pomeranz and S.M. Reddy, "On n-Detection Test Sequences for Synchronous Sequential Circuits," in *VLSI Test Symposium*, pp. 336–342, 1997.

[187] 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.

[188] A.K. Pramanick and S.M. Reddy, "On the Detection of Delay Faults," in *International Test Conference*, pp. 845–856, 1988.

[189] M.K. Prasad, P. Chong and K. Keutzer, "Why is ATPG Easy?," in *IEEE Design Automation Conference*, pp. 22–28, 1999.

[190] W. Qiu and D.M.H. Walker, "An Efficient Algorithm for Finding the K Longest Testable Paths Through Each Gate in a Combinational Circuit," in *International Test Conference*, pp. 592–601, 2003.

[191] S. Raj, S.B.K. Vrudhula and J. Wang, "A Methodology to Improve Timing Yield in the Presence of Process Variations," in *IEEE Design Automation Conference*, pp. 448–453, 2004.

[192] J. Rajski, J. Tyszer, M. Kassab and N. Mukherjee, "Embedded Deterministic Test," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, pp. 776–792, May 2004.

[193] T.R.N. Rao and E. Fujiwara, *Error-Control Coding for Computer Systems*. Prentice Hall, January 1989.

[194] K. Ravi and F. Somenzi, "Minimal Assignments for Bounded Model Checking," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 2988, pp. 31–45, Springer, 2004.

[195] J. Rearick and J.H. Patel, "Fast and Accurate CMOS Bridging Fault Simulation," in *International Test Conference*, pp. 54–62, 1993.

[196] S.M. Reddy, *Models in Hardware Testing*, Chapter 3. Springer, 2010.

[197] M. Renovell, F. Azaïs and Y. Bertrand, "Detection of Defects Using Fault Model Oriented Test Sequences," *Journal of Electronic Testing*, vol. 14, pp. 13–22, 1999.

[198] M. Renovell, P. Huc and Y. Bertrand, "CMOS Bridge Fault Modeling," in *VLSI Test Symposium*, pp. 392–397, 1994.

[199] M. Renovell, P. Huc and Y. Bertrand, "The Concept of Resistance Interval: A New Parametric Model for Resistive Bridging Fault," in *VLSI Test Symposium*, pp. 184–189, 1995.

[200] R. Rodríguez-Montañés, E.M.J.G. Bruls and J. Figueras, "Bridging Defects Resistance Measurements in a CMOS Process," in *International Test Conference*, pp. 892–899, 1992.

[201] J.P. Roth, "Diagnosis of Automata Failures: A Calculus and a Method," *IBM Journal of Research and Development*, vol. 10, pp. 278–281, July 1966.

[202] J.P. Roth, W.G. Bouricius and P.R. Schneider, "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits," *IEEE Transactions on Electronic Computers*, vol. 16, pp. 567–579, October 1967.

[203] 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.

[204] Y. Sato, I. Yamazaki, H. Yamanaka, T. Ikeda and M. Takakura, "A Persistent Diagnostic Technique for Unstable Defects," in *International Test Conference*, pp. 242–249, 2002.

[205] M. Sauer, A. Czutro, B. Becker and I. Polian, "On the Quality of Test Vectors for Post-Silicon Characterization," in *IEEE European Test Symposium*, May 2012.

[206] M. Sauer, A. Czutro, I. Polian and B. Becker, "Estimation of Component Criticality in Early Design Steps," in *IEEE International On-line Testing Symposium*, pp. 104–110, July 2011.

[207] M. Sauer, A. Czutro, I. Polian and B. Becker, "Small-Delay-Fault ATPG with Waveform Accuracy," in *International Conference on Computer-Aided Design*, pp. 30–36, November 2012.

[208] M. Sauer, A. Czutro, T. Schubert, S. Hillebrecht, I. Polian and B. Becker, "SAT-Based Analysis of Sensitisable Paths," in *IEEE Design and Diagnostics of Electronic Circuits and Systems*, pp. 93–98, April 2011. Best Paper Award in the Test Category.

[209] M. Sauer, J. Jiang, A. Czutro, I. Polian and B. Becker, "Efficient SAT-Based Search for Longest Sensitisable Paths," in *IEEE Asian Test Symposium*, November 2011.

[210] M. Sauer, S. Kupferschmid, A. Czutro, I. Polian, S.M. Reddy and B. Becker, "Functional Test of Small-Delay Faults using SAT and Craig Interpolation," in *International Test Conference*, pp. 1–8, November 2012.

[211] M. Sauer, S. Kupferschmid, A. Czutro, S.M. Reddy and B. Becker, "Analysis of Reachable Sensitisable Paths in Sequential Circuits with SAT and Craig Interpolation," in *International Conference on VLSI Design*, January 2012.

[212] M. Sauer, S. Reimer, I. Polian, T. Schubert and B. Becker, "Provably Optimal Test Cube Generation Using Quantified Boolean Formula Solving," in *ASP Design Automation Conference*, 2013.

[213] M. Sauer, S. Reimer, T. Schubert, I. Polian and B. Becker, "Efficient SAT-Based Dynamic Compaction and Relaxation for Longest Sensitizable Paths," in *Conference on Design, Automation and Test in Europe*, 2013.

[214] J. Savir and S. Patil, "Scan-Based Transition Test," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, pp. 1232–1241, August 1993.

[215] J. Savir and S. Patil, "Broad-side Delay Test," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, pp. 1057–1064, August 1994.

[216] R.R. Schneider, "On the Necessity to Examine D-Chains in Diagnostic Test Generation," *IBM Journal of Research and Development*, vol. 11, p. 114, January 1967.

[217] T. Schubert, M. Lewis and B. Becker, "antom — Solver Description," in *SAT Race*, 2010.

[218] M.H. Schulz and E. Auth, "Improved Deterministic Test Pattern Generation with Application to Redundancy Identification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, no. 7, pp. 811–816, 1989.

[219] M.H. Schulz, E. Trischler and T.M. Sarfert, "SOCRATES: A Highly Efficient Automatic Test Pattern Generation System," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 7, pp. 126–137, January 1988.

[220] F.F. Sellers, M.Y. Hsiao and L.W. Bearnson, "Analyzing Errors With the Boolean Difference," *IEEE Transactions on Computers*, vol. 17, pp. 676–683, July 1968.

[221] B. Selman, H.A. Kautz and B. Cohen, "Noise Strategies for Local Search," in *AAAI National Conference on Artificial Intelligence*, 1994.

[222] Y. Shao, S.M. Reddy, I. Pomeranz and S. Kajihara, "On Selecting Testable Paths in Scan Designs," *Journal of Electronic Testing*, vol. 19, no. 4, pp. 447–456, 2003.

[223] M. Sheeran and G. Stålmarck, "A Tutorial on Stålmarck's Proof Procedure for Propositional Logic," in *International Conference on Formal Methods in CAD*, pp. 82–99, Springer, 1998.

[224] J. Shi, G. Fey, R. Drechsler, A. Glowatz, F. Hapke and J. Schlöffel, "PASSAT: Efficient SAT-Based Test Pattern Generation for Industrial Circuits," in *IEEE International Symposium on VLSI*, pp. 212–217, 2005.

[225] T. Shinogi, T. Kanbayashi, T. Yoshikawa, S. Tsuruoka and T. Hayashi, "Faulty Resistance Sectioning Technique for Resistive Bridging Fault ATPG Systems," in *IEEE Asian Test Symposium*, pp. 76–81, 2001.

[226] O. Shtrichman, "Pruning Techniques for the SAT-Based Bounded Model Checking Problem," *Lecture Notes in Computer Science*, vol. 2144, pp. 58–70, 2001.

[227] D.P. Siewiorek and R.S. Swarz, *Reliable Computer Systems — Design and Evaluation*. Digital Press, 1992.

[228] A. Sinha, S.K. Gupta and M.A. Breuer, "Validation and Test Generation for Oscillatory Noise in VLSI Interconnects," in *IEEE International Conference on Computer-Aided Design*, pp. 289–296, 1999.

[229] A. Sinha, S.K. Gupta and M.A. Breuer, "Validation and Test Issues Related to Noise Induced by Parasitic Inductances of VLSI Interconnects," *IEEE Transactions Advanced Packaging*, vol. 25, no. 3, pp. 329–339, 2002.

[230] J.K. Slaney and T. Walsh, "Phase Transition Behavior: From Decision to Optimization," in *International Symposium on Theory and Applications of Satisfiability Testing*, 2002.

[231] G.L. Smith, "Model for Delay Faults Based upon Paths," in *International Test Conference*, pp. 342–349, 1985.

[232] J.E. Smith and G. Metze, "Strongly Fault Secure Logic Networks," *IEEE Transactions on Computers*, vol. 27, no. 6, pp. 491–499, 1978.

[233] T.J. Snethen, "Simulator-Oriented Fault Test Generator," in *IEEE Design Automation Conference*, pp. 88–93, June 1977.

[234] S. Spinner, I. Polian, P. Engelke, B. Becker, M. Keim and W.-T. Cheng, "Automatic Test Pattern Generation for Interconnect Open Defects," in *VLSI Test Symposium*, pp. 181–186, 2008.

[235] T. Stanion, D. Bhattacharya and C. Sechen, "An Efficient Method for Generating Exhaustive Test Sets," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, pp. 1516–1524, December 1995.

[236] D.P. Steele, "Ground Bounce in CMOS ASICs," in *IEEE ASIC Seminar and Exhibit*, pp. 1–4, 1989.

[237] P. Stephan, R.K. Brayton and A.L. Sangiovanni-Vincentelli, "Combinational Test Generation Using Satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, pp. 1167–1176, September 1996.

[238] P. Tafertshofer and A. Ganz, "SAT Based ATPG Using Fast Justification and Propagation in the Implication Graph," in *International Conference on Computer-Aided Design*, pp. 139–146, 1999.

[239] P. Tafertshofer, A. Ganz and K.J. Antreich, "IGRAINE — An Implication GRaph-bAsed engINE for Fast Implication, Justification and Propagation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, pp. 907–927, August 2000.

[240] P. Tafertshofer, A. Ganz and M. Henftling, "A SAT-Based Implication Engine for Efficient ATPG, Equivalence Checking and Optimization of Netlists," in *International Conference on Computer-Aided Design*, pp. 648–655, 1997.

[241] D. Tille, S. Eggersglüß and R. Drechsler, "Incremental Solving Techniques for SAT-Based ATPG," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, pp. 1125–1130, July 2010.

[242] D. Tille, S. Eggersglüß, G. Fey, R. Drechsler, A. Glowatz, F. Hapke and J. Schlöffel, "Studies on Integrating SAT-Based ATPG in an Industrial Environment," in *GI/ITG Workshop "Testmethoden und Zuverlässigkeit von Schaltungen und Systemen"*, 2007.

[243] D. Tille, S. Eggersglüß, R. Krenz-Bååth, J. Schlöffel and R. Drechsler, "Improving CNF Representations in SAT-Based ATPG for Industrial Circuits Using BDDs," in *IEEE European Test Symposium*, pp. 176–181, May 2010.

[244] D. Tille, R. Krenz-Bååth, J. Schlöffel and R. Drechsler, "Improved Circuit-to-CNF Transformation for SAT-Based ATPG," in *GI/ITG Workshop "Testmethoden und Zuverlässigkeit von Schaltungen und Systemen"*, pp. 25–29, February 2008.

[245] C. Tirumurti, S. Kundu, S. Sur-Kolay and Y.-S. Chang, "A Modeling Approach for Addressing Power Supply Switching Noise Related Failures of Integrated Circuits," in *Conference on Design, Automation and Test in Europe*, 2004.

[246] G.S. Tseitin, "On the Complexity of Derivations in Propositional Calculus," in *Studies in Constructive Mathematics and Mathematical Logics* (A. Slisenko, ed.), 1968.

[247] C.-W. Tseng and E.J. McCluskey, "Multiple-Output Propagation Transition Fault Test," in *International Test Conference*, pp. 358–366, 2001.

[248] E.G. Ulrich and T.G. Baker, "Concurrent Simulation of Nearly Identical Digital Networks," *Computer*, vol. 7, pp. 39–44, April 1974.

[249] J.A. Waicukauski, E.B. Eichelberger, O.P. Forlenza, E. Lindbloom and T. McCarthy, "Fault Simulation for Structured VLSI," *VLSI Systems Design*, vol. 6, pp. 20–32, December 1985.

[250] J.A. Waicukauski, P.A. Shupe, D.J. Giramma and A. Matin, "ATPG for Ultra-Large Structured Designs," in *International Test Conference*, pp. 44–51, 1990.

[251] J. Wang, X. Lu, W. Qiu, Z. Yue, S. Fancler, W. Shi and D.M.H. Walker, "Static Compaction of Delay Tests Considering Power Supply Noise," in *VLSI Test Symposium*, pp. 235–240, 2005.

[252] J. Wang, Z. Yue, X. Lu, W. Qiu, W. Shi and D.M.H. Walker, "A Vector-Based Approach for Power Supply Noise Analysis in Test Compaction," in *International Test Conference*, 2005.

[253] L.-T. Wang, C.-W. Wu and X. Wen, *VLSI Test Principles and Architectures*. Morgan Kaufmann, 2006.

[254] F. Wanlass, "Low Stand-by Power Complementary Field Effect Circuitry," 1967. United States patent 3,356,858.

[255] J. Whittemore, J. Kim and K.A. Sakallah, "SATIRE: A New Incremental Satisfiability Engine," in *IEEE Design Automation Conference*, pp. 542–545, June 2001.

[256] T.W. Williams and K.P. Parker, "Testing Logic Networks and Designing for Testability," *IEEE Computer*, vol. 12, pp. 9–21, October 1979.

[257] M. Yilmaz, K. Chakrabarty and M. Tehranipoor, "Test-Pattern Selection for Screening Small-Delay Defects in Very-Deep Submicrometer Integrated Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, pp. 760–773, May 2010.

[258] H. Zhang, "SATO: An Efficient Propositional Prover," in *International Conference on Automated Deduction*, pp. 155–160, July 1997.

[259] L. Zhang, C.F. Madigan, M.W. Moskewicz and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver," in *International Conference on Computer-Aided Design*, November 2001.

[260] M. Zhang, S. Mitra, T.M. Mak, N. Seifert, N.J. Wang, Q. Shi, K.S. Kim, N. R. Shanbhag and S.J. Patel, "Sequential Element Design With Built-In Soft Error Resilience," *IEEE Transactions on VLSI Systems*, vol. 14, no. 12, pp. 1368–1378, 2006.

[261] Y. Zhao and S. Dey, "Analysis of Interconnect Crosstalk Defect Coverage of Test Sets," in *International Test Conference*, pp. 492–501, 2000.

[262] C.G. Zoellin, H.-J. Wunderlich, I. Polian and B. Becker, "Selective Hardening in Early Design Steps," in *IEEE European Test Symposium*, pp. 185–190, 2008.

[263] C. Ökmen, M. Keim, R. Krieger and B. Becker, "On Optimizing BIST Architecture by Using OBDD-based Approaches and Genetic Algorithms," in *VLSI Test Symposium*, pp. 426–431, 1997.