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

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* $\textsc{cv}(g)$.

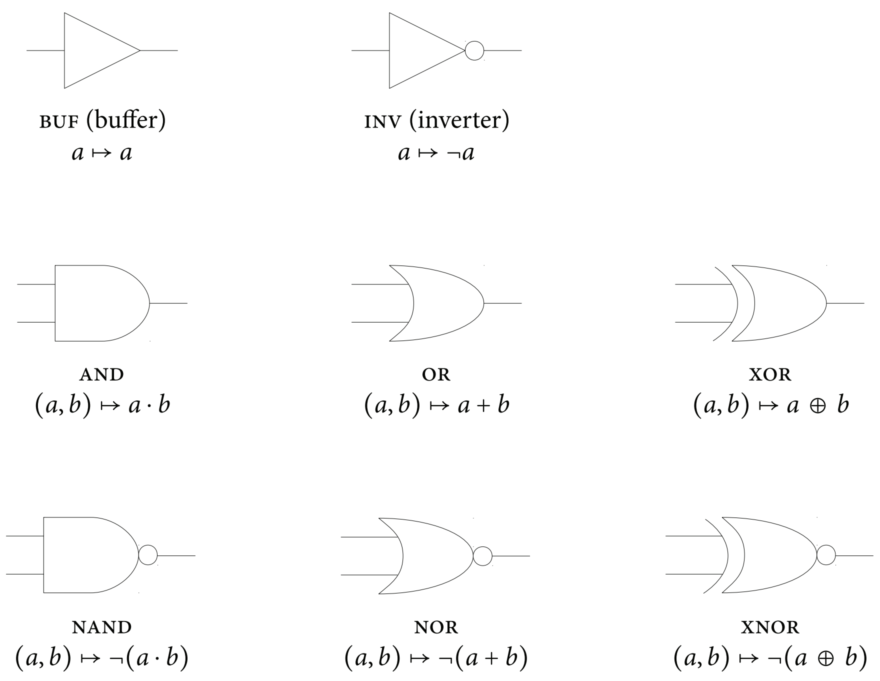

BUF (buffer)
$a \mapsto a$
INV (inverter)
$a \mapsto \neg a$
AND
$(a, b) \mapsto a \cdot b$
OR
$(a, b) \mapsto a + b$
XOR
$(a, b) \mapsto a \oplus b$
NAND
$(a, b) \mapsto \neg(a \cdot b)$
NOR
$(a, b) \mapsto \neg(a + b)$
XNOR
$(a, b) \mapsto \neg(a \oplus b)$

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.

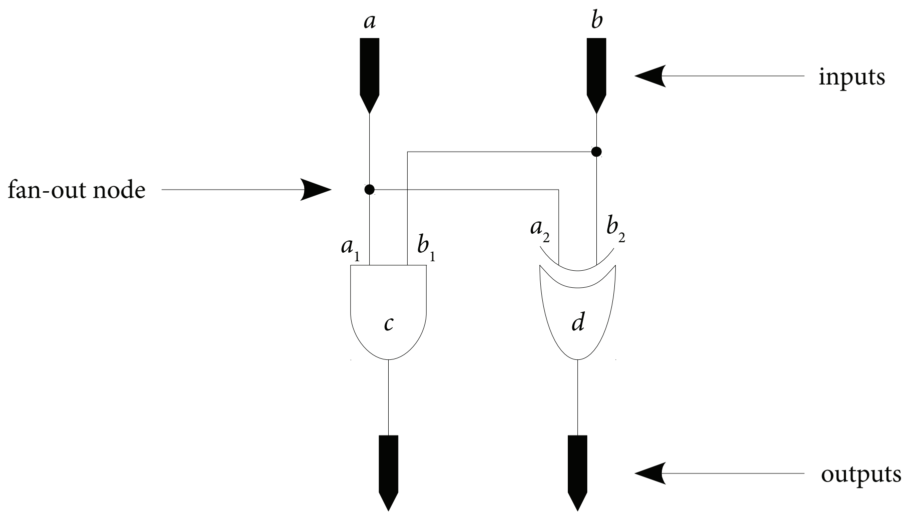


FIGURE 3. GATE-LEVEL HALF-ADDER

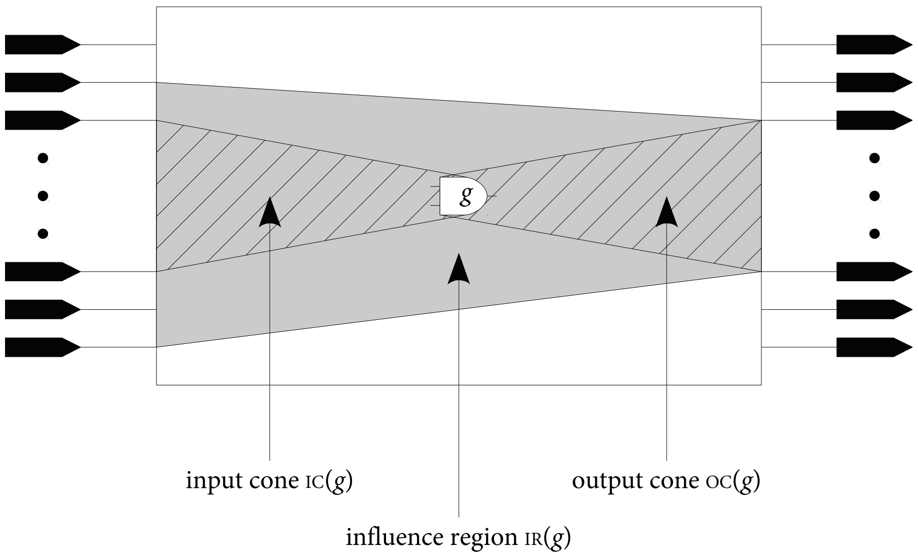


FIGURE 4. CONES OF INFLUENCE

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.

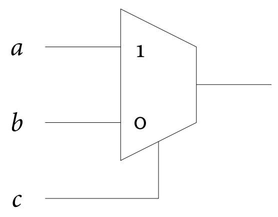
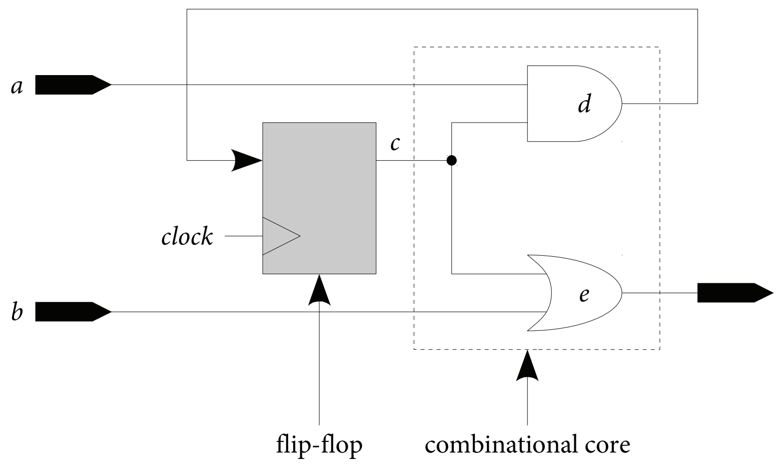


FIGURE 5. A TWO-INPUT MULTIPLEXER

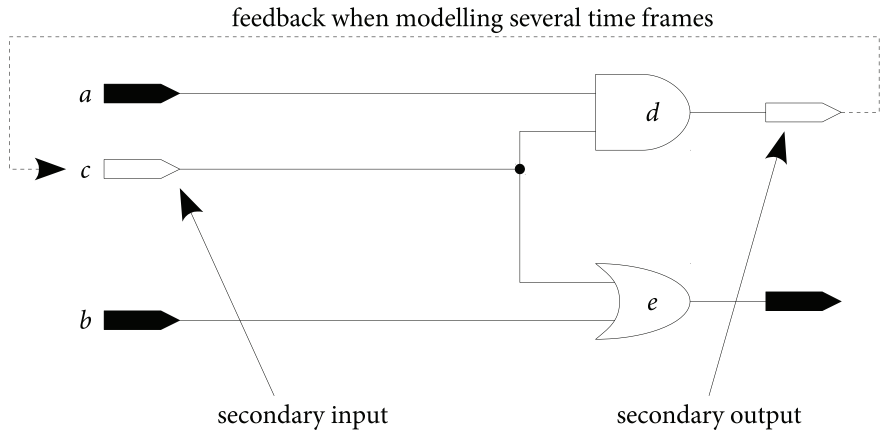
To close this section, Figure 5 shows the symbol used in this thesis to represent *multiplexers*. A two-input multiplexer is a circuit that implements the Boolean function $(a, b, c) \mapsto (c \cdot a) + (\neg c \cdot b)$, i.e. the control input c selects which of the two inputs a and b is passed to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

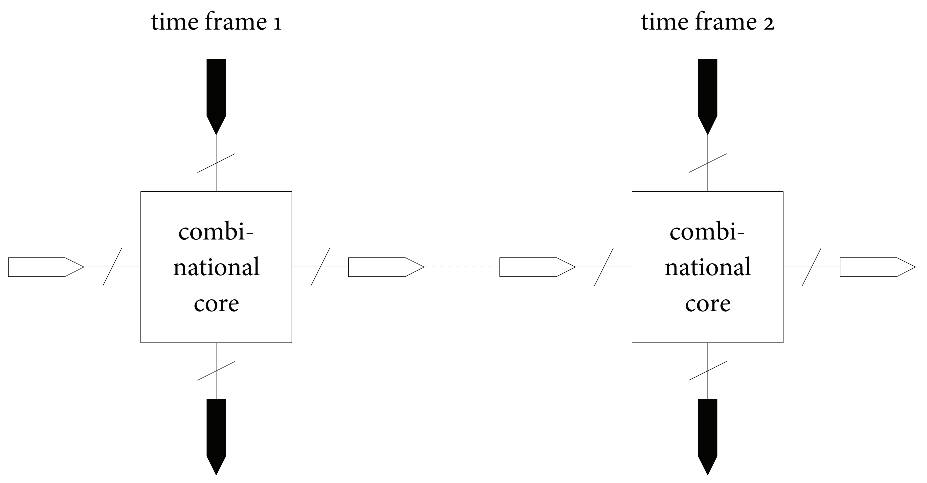


Figure 7. Sequential expansion

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* ($0\rightarrow1$) or *falling* ($1\rightarrow0$) 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].

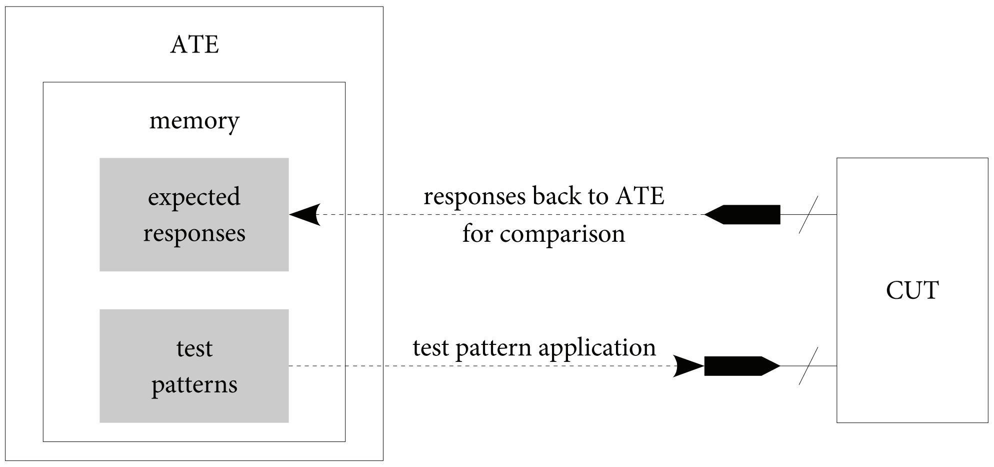


Figure 8. Test application

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 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.

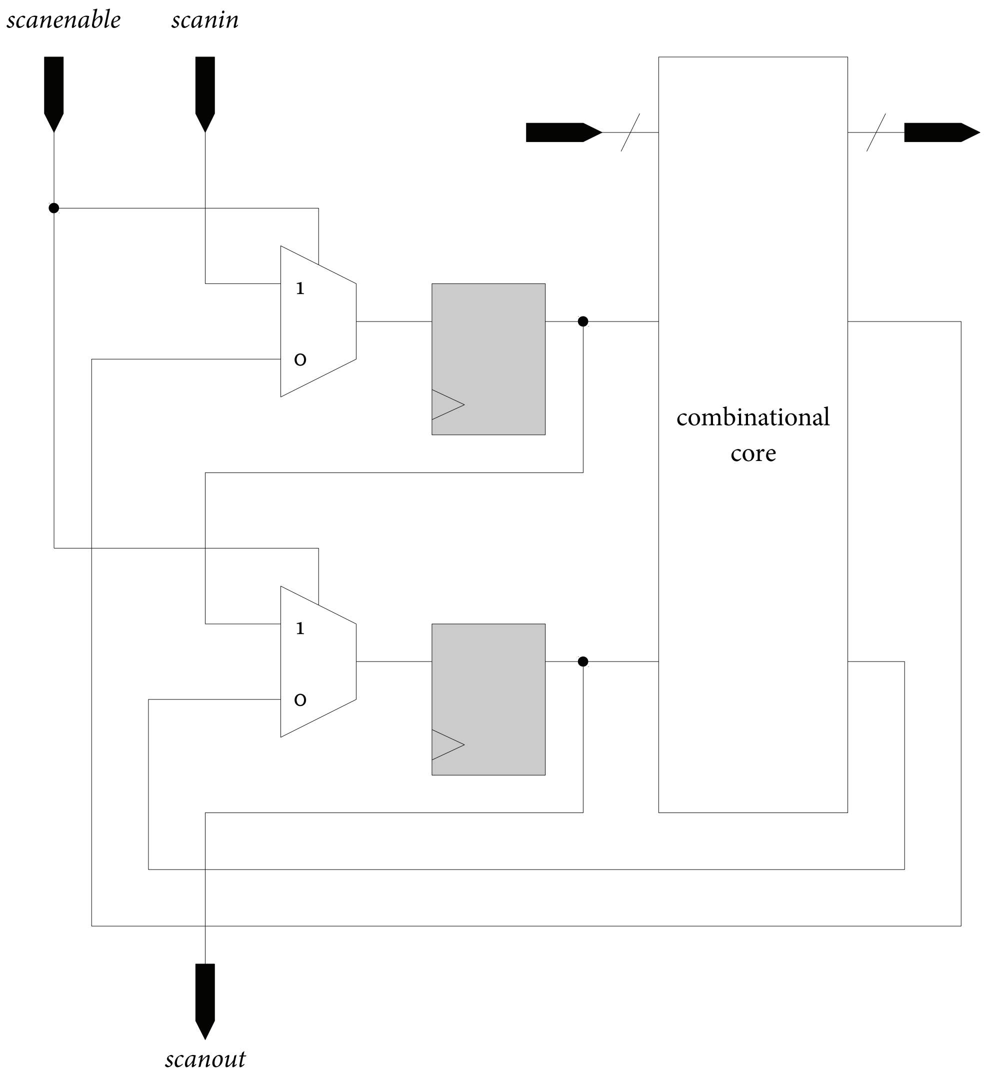


FIGURE 9. SCAN DESIGN

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.

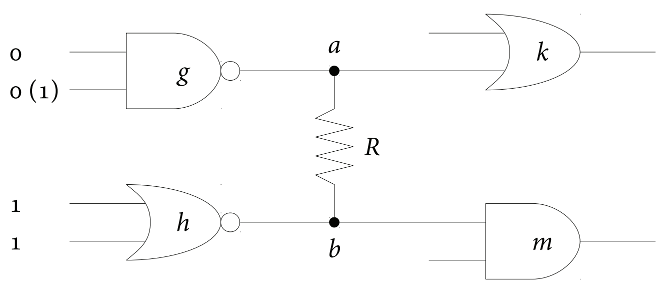
2.5 Resistive fault models

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].

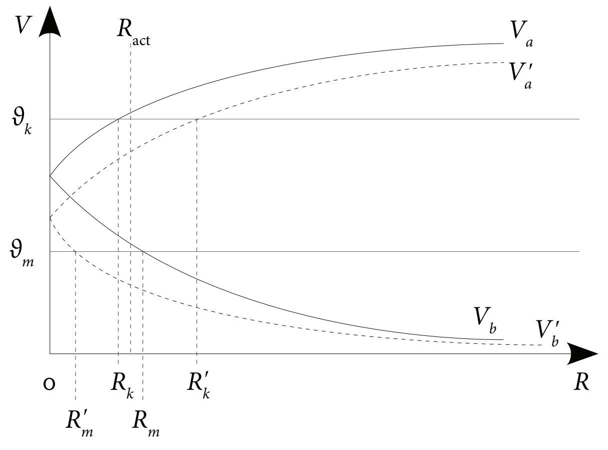
However, these two bridging models are too simple to reflect the behaviour of realistic short defects. Hence, more advanced fault models have been proposed [15, 23, 85, 84, 167, 195], but these modelling approaches disregard the fact that a substantial fraction of short defects are resistive [200] and 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

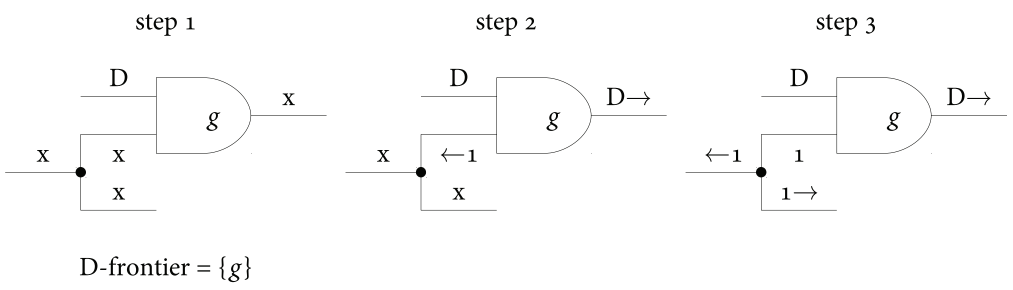


FIGURE 11. D-ALGORITHM — AN EXAMPLE CHAIN OF IMPLICATIONS

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*,
- $(\varphi \land \psi)$ — *conjunction*,
- $(\varphi \lor \psi)$ — *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 $((\varphi\land\neg\psi)\lor(\psi\land\neg\varphi))$,
- $(\varphi \to \psi)$ — *implication* — as an abbreviation of $(\neg\varphi \lor \psi)$,
- $(\varphi \leftrightarrow \psi)$ — *equivalence* — as an abbreviation of $((\varphi \to \psi) \land (\psi \to \varphi))$,
- $(\varphi_1 \lor \varphi_2 \lor \varphi_3)$ — as an abbreviation of $((\varphi_1 \lor \varphi_2) \lor \varphi_3)$ or of $(\varphi_1 \lor (\varphi_2 \lor \varphi_3))$,
- $(\varphi_1 \land \varphi_2 \land \varphi_3)$ — as an abbreviation of $((\varphi_1 \land \varphi_2) \land \varphi_3)$ or of $(\varphi_1 \land (\varphi_2 \land \varphi_3))$.

Let $\mathfrak{X}$ be the set of variables occurring in a formula φ. A *logic assignment* or *Boolean assignment* is a map $w : \mathfrak{X} \to \mathbb{B}$ 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:

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

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 $\mathbb{B}^n \to \mathbb{B}$ 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*
DPLL($\varphi$) {
    for each unit clause $K$ in $\varphi$ do {
        $\varphi$ := APPLY-UNIT-PROPAGATION($\varphi, K$)
    }
    for each pure literal $\lambda$ in $\varphi$ do {
        $\varphi$ := APPLY-PURE-LITERAL-RULE($\varphi, \lambda$)
    }
    if $\varphi$ is empty then {
        return *satisfiable*
    }
    if $\varphi$ contains the empty clause then {
        return *unsatisfiable*
    }
    $X$ := SELECT-VARIABLE($\varphi$)
    if DPLL($\varphi \wedge X$) returns *satisfiable* then {    ▷ decision ($X = 1$)
        return *satisfiable*
    } else {
        return DPLL($\varphi \wedge \neg X$)                 ▷ backtracking
    }
}
```

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].

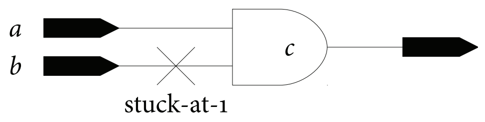
Given a combinational circuit C with n inputs and m outputs, SAT-based test generation for a fault f consists of three steps: First, fault f is injected into the original circuit, which renders a faulty version C^f of the circuit. Then, the original circuit and the faulty version are combined to form 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 $i = 1, \dots, n$, while the j-th output of C and the j-th output of C^f are both connected to a new XOR gate for $j = 1, \dots, m$. 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.

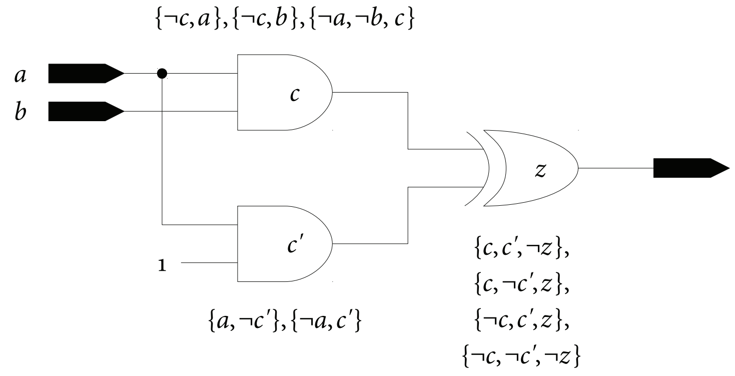
Finally, the last step consists in solving the generated SAT instance using a SAT solver. If the SAT solver finds a solution, the test pattern is composed of the values that have been assigned to the Boolean variables that represent the primary inputs.

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

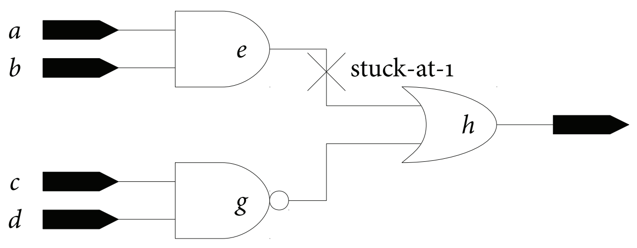
$$\{\{\neg c, a\}, \{\neg c, b\}, \{\neg a, \neg b, c\}\}.$$

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

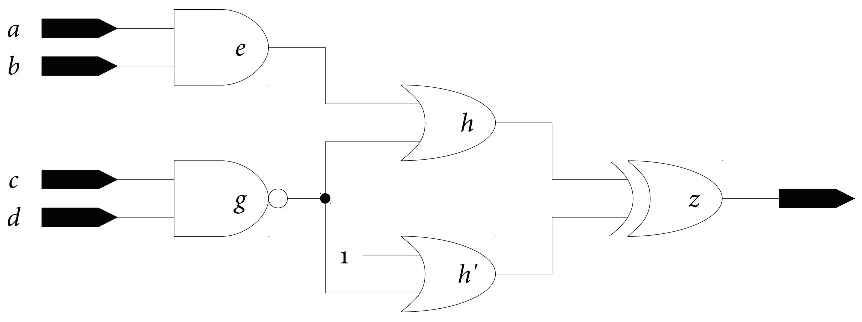
$$(a, b, c, c', z) \mapsto (1, 0, 0, 1, 1).$$

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.