Applying Advanced SAT-Based Techniques to Circuit Testing

Dissertation zur Erlangung des Doktorgrades
der Technischen Fakultät
der Albert-Ludwigs-Universität
Freiburg

vorgelegt von

Jan Burchard

Albert-Ludwigs-Universität Freiburg

Dean:	Prof. Dr. Oliver Paul, Albert-Ludwigs-Universität Freiburg, Germany
First Co-chair:	Prof. Dr. Bernd Becker, Albert-Ludwigs-Universität Freiburg, Germany
Second Co-chair:	Prof. Dr. Krishnendu Chakrabarty Duke University, United States of America
Examination Date:	April 20, 2018

Zusammenfassung

In integrierten Schaltungen ist das Auftreten von Herstellungsdefekten eher Regel als Ausnahme. Dies wird durch das kontinuierliche Schrumpfen der Strukturgrößen und das gleichzeitige Wachsen der Anzahl an Transistoren pro Chip noch verstärkt. Um trotzdem eine hohe Qualität zu gewährleisten ist es unerlässlich einen Chip nach der Fertigung umfangreich zu testen.

Da ein vollständiger funktionaler Test aufgrund der enormen Anzahl an Eingangsbelegungen nicht möglich ist, werden digitale Schaltkreise im Allgffemeinen strukturell mit Fehlermodellen getestet. Ein Fehlermodell abstrahiert von der großen Gesamtzahl möglicher Defekte auf eine kleinere Menge von modellierten Fehlern. Die modellierten Fehler haben das Ziel die Auswirkungen von häufig auftretenden Defekten abzudecken. Die Abwesenheit dieser Fehler wird anschließend über speziell generierte Testmuster überprüft. Nur durch passende Fehlermodelle kann dabei auch eine hohe Defektabdeckung erreicht werden.

Mit der eingangs diskutierten steigenden Integrationsdichte und Transistoranzahl moderner Schaltkreise steigt auch das Potential für mögliche Defekte. Aus diesem Grund reichen einfachere Fehlermodelle oft nicht mehr aus, um einen Großteil der defekten Chips zu erkennen. Daher werden neue Ansätze, die komplexe elektrische Vorgänge effizient modellieren können, zur Erstellung von Testmustern für moderne Schaltkreise immer essentieller. Ein vielversprechender Ansatz sind SAT-basierte Methoden, die das Problem zunächst formal als Boolesche Formel beschreiben um diese dann mit einem spezialisierten SAT-Solver zu lösen. Dies erlaubt es die Frage nach der Existenz eines Testmusters auf die Lösbarkeit einer Formel zu reduzieren. Aus einer erfüllenden Belegung der Formel kann hiernach ein Testmuster hergeleitet werden.

In dieser Dissertation werden drei wichtige Beiträge zum SAT-basierten Testen von Schaltkreisen vorgestellt.

Der erste Teil dieser Arbeit befasst sich mit D-chains. D-chains erweitern die Boolesche Formel, die bei einer SAT-basierten Testmustergenerierung erstellt wird, um strukturelle Informationen zur Ausbreitung eines Fehlereffektes. Diese zusätzlichen Informationen reduzieren die Größe des Suchraums und können so den Lösungsvorgang beschleunigen. Im Rahmen dieser Dissertation werden ein neues D-chain Konzept sowie zwei hybride D-chains entwickelt und vorgestellt. Darüber hinaus erfolgt zum ersten Mal ein umfassender Vergleich aller gebräuchlichen D-chain Ansätze.

Die neu entwickelte hybride D-chain mit einer dynamischen Heuristik zur Knotenauswahl zeigt in der Evaluation den größten Geschwindigkeitsgewinn aller betrachteter D-chain Varianten. Sie verringert die Zeit zur Lösungsberechnung um durchschnittlich 74 % im Vergleich zu der Variante ohne D-chains. Damit sind die neuen D-chains den bekannten Ansätzen überlegen und bieten die beste Beschleunigung für die SAT-basierte Testmustergenerierung.

Im zweiten Teil dieser Dissertation wird die SAT-basierte Testmustergenerierung selbst vorangetrieben. Aktuelle Forschungsergebnisse zeigen, dass das Testen von Logikzellen immer wichtiger wird, da Defekte innerhalb einer Zelle von klassischen Fehlermodellen nicht zuverlässig entdeckt werden. Das Transistor stuck-open Fehlermodell bietet hier eine vielversprechende Lösung. Es modelliert den Fall, dass ein einzelner Transistor innerhalb einer Zelle dauerhaft nicht leitet und deckt damit sowohl Defekte, die den Transistor selbst betreffen, als auch Defekte in den Leitungen ab.

Die Testmustergenerierung für einen Transistor stuck-open Fehler wird von zwei Effekten erschwert, die für eine hohe Genauigkeit beachtet werden sollten. Eine erfolgreiche Fehlerentdeckung kann ansonsten nicht garantiert werden. Zum Einen muss sichergestellt werden, dass bestimmte Eingänge der fehlerhaften Logikzelle stabil und frei von Störimpulsen (Glitches) sind, da der Fehlereffekt sonst maskiert werden könnte. Darüber hinaus kann der Fehlereffekt auch durch das sogenannte charge-sharing maskiert werden. Beim charge-sharing wird die Ladung innerhalb der Logikzelle zwischen verschiedenen Leitungen verteilt, wodurch sich die Spannung am Ausgang der Zelle möglicherweise signifikant verändert.

Der in dieser Dissertation vorgestellte deterministische Algorithmus zur Generierung von Testmustern für das Transistor stuck-open Modell garantiert sowohl deren Glitch-Freiheit, als auch die Abwesenheit von charge-sharing Konflikten. Darüber hinaus unterstützt er die Generierung von Testmustern, die gezielt Glitches einsetzen um einen Fehler zu aktivieren oder einen charge-sharing Konflikt zu umgehen.

Die ausführliche Evaluierung der vorgestellten Methode zeigt deutlich, dass Glitches beim Testen von Transistor stuck-open Fehlern unbedingt beachtet werden müssen. Ist dies nicht der Fall sind in den durchgeführten Experimenten durchschnittlich über 25 % der erstellten Testmuster ungültig. Das eingeführte Verfahren dagegen kann selbst für große Schaltkreise gültige Testmuster erstellen die frei von Glitches und charge-sharing Konflikten sind. Darüber hinaus kann der Algorithmus die Fehlerabdeckung durch das gezielte Ausnutzen von Glitches noch weiter erhöhen und so auch Fehler testen, die von einem konventionellen Ansatz als untestbar klassifiziert werden.

Der letzte Teil dieser Dissertation behandelt die Charakterisierung von *möglicherweise erkannten* Fehlern. Ein Fehler wird als möglicherweise erkannt klassifiziert, wenn seine Detektion von der Belegung an unbekannten oder unkontrollierbaren Eingängen abhängt. Der entwickelte Ansatz erlaubt es die Wahrscheinlichkeit für die Fehlerentdeckung exakt zu berechnen. Dies ermöglicht die genaue Bestimmung der Gesamtfehlerabdeckung des Schaltkreises. Die gewonnenen Informationen können weiterhin bei Entscheidungen zur Erhöhung der Fehlerabdeckung genutzt werden, zum Beispiel um Testpunkte einzufügen.

Das vorgestellte Verfahren modelliert die Charakterisierung der möglicherweise erkannten Fehler als #SAT-Problem. Dabei wird die Anzahl der erfüllenden Belegungen einer Booleschen Formel mit einem #SAT-Solver berechnet. Aus dieser Anzahl lässt sich die Entdeckungswahrscheinlichkeit für den aktuell betrachteten möglicherweise erkannten Fehler herleiten.

Die Evaluation des präsentierten Ansatzes ergibt, dass ein Großteil der möglicherweise erkannten Fehler akkurat charakterisiert werden kann. Die Ergebnisse zeigen, dass die Entdeckungswahrscheinlichkeit stark vom Schaltkreis und der Eingangsbelegung abhängt und nicht mit einem fixen Wert abgeschätzt werden kann. Der vorgestellte Algorithmus liefert daher einen klaren Mehrwert über den aktuellen Stand der Technik hinaus.

Das #SAT-Problem – die Berechnung der Anzahl der erfüllenden Belegungen einer Booleschen Formel – ist sehr schwer zu lösen. Um das entwickelte Verfahren zur Charakterisierung von möglicherweise erkannten Fehlern auch auf große Schaltkreise anwenden zu können, präsentiert diese Dissertation außerdem den verteilten #SAT-Solver `dCountAntom`. Dieser setzt CPU-Kerne auf verschiedenen Computern gleichzeitig ein und verbessert die Skalierbarkeit dadurch signifikant. Außerdem wird ein Verfahren zur frühen Abschätzung der Gesamtlösungszeit vorgestellt. Basierend auf diesem Verfahren ist es möglich den Lösungsvorgang für zu schwere Formeln frühzeitig abzubrechen. Die experimentellen Ergebnisse für Formeln die unterschiedliche realistische Probleme modellieren zeigen eine große Beschleunigung des Lösungsvorgangs wenn viele CPU-Kerne eingesetzt werden.

Contents

1. Introduction

Manufacturing defects in integrated circuits have always been a problem, especially with very large scale integration. This problem is further fueled by the progressively smaller feature sizes and, at the same time, a growing number of transistors per chip. Thus, to ensure a high quality of the shipped devices a thorough testing is necessary.

Since a complete functional test is generally impossible because of the huge number of possible input assignments, digital circuits are usually tested structurally based on fault models. A fault model abstracts from the high number of possible defects to a smaller number of modeled faults. The aim of the fault model is to cover the most probable defects. Only a fitting fault model can therefore ensure a high defect coverage. The absence of the modeled faults is then checked with specifically crafted test patterns.

With the above mentioned increase in integration density and the vast quantity of transistors in modern circuits, the potential for defects is growing. For this reason, basic fault models (e.g., the stuck-at fault model) are often not sufficient to detect all of the defects and, therewith, the majority of defective devices. More complex fault models that are more closely related to real defects can be used to increase the defect coverage. However, the generation of test patterns for these fault models is often more difficult. As a result, novel approaches for the generation of test patterns which can efficiently model complex electrical properties are becoming more and more relevant. SAT-based methods, which first formally describe the problem as a Boolean formula before solving it with a specialized SAT solver, are a promising approach in that direction. With these approaches the problem of finding a test pattern for a fault is reduced to the satisfiability of a formula. A test pattern for the fault can be derived from a satisfying assignment of the formula.

1.1. Contributions

In this thesis three major contributions to the area of SAT-based testing are presented.

The first contribution focuses on the development and analysis of D-chains. D-chains extend the Boolean formula that is utilized by a SAT-based test pattern generation approach with structural information regarding the fault propagation. This information reduces the size of the search space that has to be covered by the SAT solver and can, therefore, increase the solving speed. In the scope of this thesis, an efficient D-chain concept and two novel hybrid D-chains that combine the new concept with an established encoding are introduced. Furthermore, the first comprehensive analysis of the different common D-chains known from literature is performed.

The evaluation shows that the newly developed hybrid D-chain with a dynamic node selection heuristic gives the largest increase in solving speed of all of the analyzed D-chain variants. On average it reduces the time to compute a solution by about 74 % in comparison to not utilizing a D-chain at all. Overall, the presented novel D-chains outperform the known approaches and offer the largest speedup for the SAT-based test pattern generation.

The second part of this thesis advances the SAT-based test pattern generation itself. Recent research results show that testing logic cells is becoming more and more important because cell-internal defects are not reliably detected by classic fault models. In this context, the transistor stuck-open fault model offers a promising solution. It models the case of a single transistor within a cell which is never conducting. It, thereby, covers defects within the transistor itself as well as some defects on the interconnects.

Generating a valid test pattern for a transistor stuck-open fault is challenging because of two effects that have to be considered to perform a high quality test. If these effects are ignored, a successful test cannot be guaranteed. On the one hand, it must be ensured that specific inputs of the faulty cell are stable and free from glitches. Otherwise the fault effect might be masked. On the other hand, the fault effect could also be masked by the so-called *charge-sharing*. Charge-sharing describes the sharing of the residual charges of different lines within the cell. This could have a significant impact on the voltage at the cell's output.

The deterministic algorithm for the generation of test patterns for the transistor stuck-open fault model presented in this work guarantees both glitch freedom as well as the absence of any charge-sharing conflicts. Furthermore, it supports the generation of test patterns that utilize glitches to initialize a fault or to mitigate a charge-sharing conflict.

The extensive evaluation of the proposed methodology clearly shows the importance of considering glitches when testing for transistor stuck-open faults. In the performed experiments on average more than 25 % of the test patterns that are generated without glitch-awareness might be invalidated. The presented approach, however, is able to generate valid test patterns that are free from dangerous glitches and charge-sharing conflicts even for large circuits. In addition, the algorithm is capable of further increasing the fault coverage by utilizing glitches. This allows for the testing of faults which would be classified as untestable by a conventional approach.

The final part of this thesis deals with the characterization of *possibly detected* faults. When the detection of a fault depends on the assignment of unknown or not controllable circuit inputs, it is classified as possibly detected. The developed approach is able to accurately compute the detection probability for these faults. This allows for an exact determination of the overall fault coverage of a circuit. Furthermore, the obtained information could be used to increase the fault coverage, for example by inserting test points that ensure the detection of faults with a low detection probability.

The presented algorithm models the characterization of potentially detected faults as a #SAT problem. To this end, the number of satisfying assignments of a generated Boolean formula is counted with a #SAT solver. From the number of satisfying assignments the detection probability of the currently considered fault can be derived.

The evaluation of the approach shows that most of the possibly detected faults can be accurately characterized. The results reveal that the detection probability of the faults is highly dependent on the circuit and the input assignment. Therefore, the approach of state-of-the art commercial tools which simply estimate the detection probability with a global user selectable value is inaccurate and the proposed algorithm has a clear benefit beyond the current state-of-the art.

The #SAT problem – counting the number of satisfying assignments of a Boolean formula – is very hard to solve. To be able to apply the algorithm for the characterization of possibly detected faults to large circuits, this thesis presents the distributed #SAT solver `dCountAntom`. `dCountAntom` utilizes CPU cores across different computers in parallel and thereby significantly improves the scalability. The experimental results show a large speedup of the solve speed across formulas from different origins when many cores are used in parallel. Furthermore, an approach for the early prediction of the solve time is presented. This allows for the abortion of the solve process when the formula is too hard to solve and is used for the implementation of a novel soft timeout mechanism.

1.2. Structure

This thesis is structured as follows: Chapter 2 introduces the basic concepts that are used in this work and focuses on formal solving methods and digital circuit testing. The subsequent chapters discuss and evaluate the different contributions of this thesis: D-chains are covered in Chapter 3. The test pattern generation for transistor stuck-open faults is described in Chapter 4. The application of #SAT-solving to circuit test is split into two parts: Chapter 5 introduces the distributed parallel #SAT solver `dCountAntom`. Thereafter, Chapter 6 is concerned with the characterization of possibly detected faults. Chapter 7 concludes this thesis with a summary.

Further information on the experimental setup, especially with regard to the used circuits, can be found in Appendix A. Appendix B gives a complete list of the author's publications.

1.3. List of Discussed Papers

This thesis is based in parts on the following previous publications by the author. For improved readability, citations of these publications within the text are replaced by a general reference of the applicable works at the beginning of each chapter.

[J1] J. Burchard, D. Erb, S. M. Reddy, A. D. Singh, and B. Becker, "On the generation of waveform-accurate hazard and charge-sharing aware tests for transistor stuck-off faults in CMOS logic circuits", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2017. DOI: `10.1109/TCAD.2017.2772825`

[J2] P. Raiola, J. Burchard, F. Neubauer, D. Erb, and B. Becker, "Evaluating the effectiveness of D-chains in SAT-based ATPG and diagnostic TPG", *Journal of Electronic Testing: Theory and Applications (JETTA)*, 2017. DOI: `10.1007/s10836-017-5693-6`

[C1] J. Burchard, F. Neubauer, P. Raiola, D. Erb, and B. Becker, "Evaluating the effectiveness of D-chains in SAT-based ATPG", in *18th IEEE Latin American Test Symposium (LATS)*, 2017. DOI: `10.1109/LATW.2017.7906752`

[C2] J. Burchard, D. Erb, A. D. Singh, S. M. Reddy, and B. Becker, "Fast and waveform-accurate hazard-aware SAT-based TSOF ATPG", in *Design, Automation and Test in Europe (DATE), 2017*, Best Paper Award in the Test Category, 2017. DOI: `10.23919/DATE.2017.7927027`

[C3] J. Burchard, D. Erb, S. M. Reddy, A. D. Singh, and B. Becker, "Efficient SAT-based generation of hazard-activated TSOF tests", in *IEEE 35th VLSI Test Symposium (VTS)*, 2017. DOI: `10.1109/VTS.2017.7928943`

[C4] J. Burchard, T. Schubert, and B. Becker, "Laissez-faire caching for parallel #SAT solving", in *SAT 2015*, ser. Lecture Notes in Computer Science, Springer International Publishing, 2015. DOI: `10.1007/978-3-319-24318-4_5`

[C5] J. Burchard, T. Schubert, and B. Becker, "Distributed parallel #SAT solving", in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2016. DOI: `10.1109/CLUSTER.2016.20`

[C6] J. Burchard, D. Erb, and B. Becker, "Characterization of possibly detected faults by accurately computing their detection probability", in *Design, Automation and Test in Europe (DATE), 2018*, 2018. DOI: `10.23919/DATE.2018.8342040`

2. Preliminaries

This chapter introduces the basic concepts that are used throughout this work. These can be grouped into two areas: formal solving methods with a focus on SAT and #SAT are introduced in Section 2.1. Circuit testing including an introduction to digital circuits is covered in Section 2.2.

Since these topics are vast fields of study by themselves, only the parts relevant to this thesis will be discussed. For more details on a specific subject, the interested reader can find additional information in the referenced publications.

For improved readability, the descriptions and definitions are sometimes simplified but always mathematically sound. More in-depth information on formal methods can be found in [1] while [2] covers a wide range of topics related to circuit test.

2.1. Formal Solving Methods

In computer science and related fields, there are many problems that can generally only be solved with a substantial amount of computational effort. A well-defined and adequately formulated problem can be solved by a mathematically sound algorithm, a formal method.

Most well-defined theoretical problems are grouped into complexity classes based on their theoretical difficulty.

In the areas covered in this thesis, the class of NP-complete problems is the most relevant. A problem is NP-complete if it is in the complexity class NP and every other problem in NP can be reduced to that problem with little effort (within polynomial-time in the problem size) [3].

Thus, if one can solve an NP-complete problem, one can solve all other problems in NP as well.

The Boolean satisfiability problem (SAT) [3] has proven to be a well suited candidate for the development of efficient formal solving methods, called solvers, because many problems in NP can be easily mapped to a Boolean formula and the corresponding decision problem.

The similarities of digital circuits that consist of logic gates implementing Boolean functions and of Boolean formulas allow for the easy formulation of many challenging problems from the realm of circuit testing as a SAT problem.

The following sections briefly introduce the background of SAT-solving and the closely related counting problem #SAT. First, Section 2.1.1 defines Boolean formulas. In Section 2.1.2 the Tseitin transformation which creates Boolean formulas in conjunctive normal form is presented. Next, Section 2.1.3 describes the Boolean satisfiability problem and the concepts of SAT-solving. Finally, Section 2.1.4 extends SAT to model counting and gives a summary of the computations performed by a #SAT solver.

2.1.1. Boolean Formulas

A Boolean (or propositional) formula Φ is built up of variables $(v_1, v_2, \dots)$ which are linked by operators. Table 2.1 gives an overview of the most common operators and their interpretation. In addition, some more advanced operators are defined in Table 2.2. These are used as syntactic sugar for added comprehensibility of complex relations.

Table 2.1.: Common operators in Boolean formulas.

Operator	Symbol	Interpretation (evaluates to *true* iff)
NOT	$\neg v_1$	$v_1 = false$
AND	$v_1 \wedge v_2$	$v_1 = true$ and $v_2 = true$
OR	$v_1 \vee v_2$	$v_1 = true$ or $v_2 = true$
XOR	$v_1 \oplus v_2$	$v_1 \neq v_2$

Table 2.2.: Advanced operators in Boolean formulas.

Operator	Symbol	Definition
Implication	$v_1 \Rightarrow v_2$	$\neg v_1 \vee v_2$
Equivalence	$v_1 \Leftrightarrow v_2$	$(v_1 \Rightarrow v_2) \wedge (v_2 \Rightarrow v_1)$

Equation 2.1 shows an example Boolean formula with the variables v_1, v_2 and v_3.

$$\Phi = \big((v_1 \vee v_2) \wedge (\neg v_1 \vee \neg v_2)\big) \vee v_3 \tag{2.1}$$

A (partial) variable assignment π assigns (some) variables a truth value (*true* or *false*). Based on a variable assignment, the formula can be evaluated to either *true* or *false*. As a shorthand $\Phi|_\pi = false$ is written if, and only if Φ evaluates to *false* for π and vice-versa for *true*. When a formula is evaluated to *true*, it is called satisfied (SAT). When it is evaluated to *false*, it is called unsatisfied. If no variable assignment π with $\Phi|_\pi = true$ exists, the formula is called unsatisfiable (UNSAT).

For the Boolean formula 2.1, the assignment

$\pi_1 = \{v_1 \rightarrow true, v_2 \rightarrow true, v_3 \rightarrow true\}$

satisfies the formula whereas

$\pi_2 = \{v_1 \rightarrow true, v_2 \rightarrow true, v_3 \rightarrow false\}$

does not.

A Boolean formula in conjunctive normal form (CNF) is composed of clauses connected by conjunctions ($\wedge$). Each clause consists of literals connected with disjunctions ($\vee$). A literal is a variable or a negated variable (v or $\neg v$). Equation 2.2 gives Formula 2.1 in an equivalent CNF.

$$\Phi_{CNF} = (v_1 \vee v_2 \vee v_3) \wedge (\neg v_1 \vee \neg v_2 \vee v_3) \tag{2.2}$$

To satisfy a Boolean formula in CNF all clauses have to be satisfied. To satisfy a clause it is sufficient that one of its literals evaluates to *true*.

Generally, transforming a Boolean formula Φ into an equivalent formula in CNF Φ_{CNF} ($\Phi \equiv \Phi_{CNF}$) could increase the formula size exponentially. This can be avoided, however, by only preserving equisatisfiability instead of complete equivalence. A formula Φ_{CNF} is equisatisfiable to Φ if Φ_{CNF} is satisfiable if, and only if, Φ is satisfiable. Equivalent formulas on the other hand are satisfied by the exact same variable assignments. In this work the Tseitin transformation [4] is used for the purpose of generating an equisatisfiable formula and will be described in detail in Section 2.1.2.

A variable that is not assigned through a partial variable assignment is called free. The concept of free variables is extended to literals: A literal is free if its variable is free. A clause with free literals which is not satisfied (yet) is called open. An open clause that has exactly one free literal l is unit. Unit clauses imply a variable assignment which makes l evaluate to $true$ because each clause has to be satisfied in order to satisfy the entire formula.

Given a partial variable assignment π the residual formula consists of the remaining open clauses of the original formula without the assigned literals.

Applying the partial variable assignment

$\pi_3 = \{v_1 \rightarrow true, v_3 \rightarrow false\}$

to Formula 2.2 satisfies the first clause and leaves the residual formula in Equation 2.3.

$$\Phi_{CNF}|_{\pi_3} = (\neg v_2) \tag{2.3}$$

The second clause of the original formula $(\neg v_1 \vee \neg v_2 \vee v_3)$ is reduced to $(\neg v_2)$ and is unit: To satisfy the formula v_2 has to be assigned to $false$.

In this thesis the term formula refers to a Boolean formula in conjunctive normal form unless otherwise noted. To increase the readability and clarity, $true$ and $false$ are replaced by 1 and 0, respectively, in many graphics.

2.1.2. Tseitin Transformation

The basic idea of the Tseitin transformation [4] is the conversion of any Boolean formula into an equisatisfiable formula in CNF which is linear in the size of the original Boolean formula. To this end, the original Boolean formula is extended with new variables. These variables are used to encode the Boolean operations within the formula into small sub-formulas in CNF. By combining all the sub-formulas, an equisatisfiable formula in CNF is created.

This process consists of two steps: Firstly, for every Boolean operator a corresponding sub-formula has to be computed. This computation has to be performed only once since the sub-formula's structure is not dependent on the encoded variables which can simply be substituted. Secondly, the initial Boolean formula is transformed into CNF by replacing every Boolean operator by the corresponding sub-formula. This adds a new variables for each operator to the resulting formula.

Computation of the Sub-Formulas

Every Boolean operator can be seen as a simple function which computes an output value based on input values. For example for the AND operator with variables v_1 and v_2 this function is $o = (v_1 \wedge v_2)$ which can be seen as the Boolean formula $o \Leftrightarrow (v_1 \wedge v_2)$. This formula can now be transformed into CNF:

$$\begin{aligned}
& o \Leftrightarrow (v_1 \wedge v_2) && (2.4)\\
\equiv\ & \big(o \Rightarrow (v_1 \wedge v_2)\big) \wedge \big((v_1 \wedge v_2) \Rightarrow o\big) && (2.5)\\
\equiv\ & \big(\neg o \vee (v_1 \wedge v_2)\big) \wedge \big(\neg(v_1 \wedge v_2) \vee o\big) && (2.6)\\
\equiv\ & \big((\neg o \vee v_1) \wedge (\neg o \vee v_2)\big) \wedge \big((\neg v_1 \vee \neg v_2) \vee o\big) && (2.7)\\
\equiv\ & (\neg o \vee v_1) \wedge (\neg o \vee v_2) \wedge (\neg v_1 \vee \neg v_2 \vee o) && (2.8)
\end{aligned}$$

Formula 2.8 is the CNF representation of the Boolean AND operator. The same calculation can be repeated for all of the different operators. The results are summarized in Table 2.3. The NOT operator is not transformed. If a negated variable occurs in an operation all of its occurrences are substituted by its negation.

Table 2.3.: The representation of the different Boolean operators as sub-formulas in CNF.

Operator	Sub-formula in CNF
AND	$(v_1 \vee \neg o) \wedge (v_2 \vee \neg o) \wedge (\neg v_1 \vee \neg v_2 \vee o)$
OR	$(\neg v_1 \vee o) \wedge (\neg v_2 \vee o) \wedge (v_1 \vee v_2 \vee \neg o)$
XOR	$(v_1 \vee v_2 \vee \neg o) \wedge (\neg v_1 \vee \neg v_2 \vee \neg o) \wedge (\neg v_1 \vee v_2 \vee o) \wedge (v_1 \vee \neg v_2 \vee o)$

Converting a Formula

To convert an entire formula into CNF through the Tseitin transformation one has to start from the innermost expression which contains only variables but no complex terms. This expression is then converted into CNF and every occurrence of the expression replaced by the output variable which is a new variable. The generated sub-formula in CNF is added to a new formula Φ_{CNF} which will be the equisatisfiable formula once the process is completed. This procedure is repeated until the entire formula has been converted and only one variable remains.

As an example consider the conversion of Formula 2.1 shown in Table 2.4. The resulting formula Φ_{CNF} consists of twelve clauses and seven variables, four of which were added through the transformation. In comparison to Formula 2.2 it is clearly larger, even though both represent Φ in CNF. However, in general a direct conversion might result in exponential growth of the formula in CNF and the Tseitin transformation gives much smaller results. This is especially true for formulas representing large circuits with many levels from the inputs to the outputs. For each logic gate the Tseitin transformation requires only a constant number of clauses. Thus, the size of the final formula is linear in the number of gates.

2.1.3. SAT-Solving

A SAT solver attempts to find a variable assignment that satisfies a given Boolean formula Φ. This satisfying assignment is also known as a model of Φ. Usually, the formula has

Table 2.4.: Converting Formula 2.1 into CNF with the Tseitin Transformation.

Step	Φ	Φ_{CNF}
Initial	$\big(\overbrace{(v_1 \vee v_2)}^{\Leftrightarrow n_1} \wedge (\neg v_1 \vee \neg v_2)\big) \vee v_3$	-
1	$\big(n_1 \wedge \overbrace{(\neg v_1 \vee \neg v_2)}^{\Leftrightarrow n_2}\big) \vee v_3$	$(\neg v_1 \vee n_1) \wedge (\neg v_2 \vee n_1) \wedge (v_1 \vee v_2 \vee \neg n_1)$
2	$\overbrace{(n_1 \wedge n_2)}^{\Leftrightarrow n_3} \vee v_3$	$(\neg v_1 \vee n_1) \wedge (\neg v_2 \vee n_1) \wedge (v_1 \vee v_2 \vee \neg n_1) \wedge (v_1 \vee n_2) \wedge (v_2 \vee n_2) \wedge (\neg v_1 \vee \neg v_2 \vee n_2)$
3	$\overbrace{n_3 \vee v_3}^{\Leftrightarrow n_4}$	$(\neg v_1 \vee n_1) \wedge (\neg v_2 \vee n_1) \wedge (v_1 \vee v_2 \vee \neg n_1) \wedge (v_1 \vee n_2) \wedge (v_2 \vee n_2) \wedge (\neg v_1 \vee \neg v_2 \vee n_2) \wedge (n_1 \vee \neg n_3) \wedge (n_2 \vee \neg n_3) \wedge (\neg n_1 \vee \neg n_2 \vee n_3)$
Final	n_4	$(\neg v_1 \vee n_1) \wedge (\neg v_2 \vee n_1) \wedge (v_1 \vee v_2 \vee \neg n_1) \wedge (v_1 \vee n_2) \wedge (v_2 \vee n_2) \wedge (\neg v_1 \vee \neg v_2 \vee n_2) \wedge (n_1 \vee \neg n_3) \wedge (n_2 \vee \neg n_3) \wedge (\neg n_1 \vee \neg n_2 \vee n_3) \wedge (\neg n_3 \vee n_4) \wedge (\neg v_3 \vee n_4) \wedge (n_3 \vee v_3 \vee \neg n_4)$

to be in CNF which simplifies the overall solver structure and allows for many of the improvements found in modern solvers.

Over the years a large number of different concepts have been applied to SAT-solving. Incomplete methods employ heuristics, for example local search, in an attempt to find a satisfying assignment [5], [6]. While these methods have shown to be fast for certain types of formulas, they cannot prove that a formula is unsatisfiable. This makes such approaches unsuitable for many applications including those pursued in this thesis because they cannot prove that a fault is untestable.

On the other hand, complete solvers can prove unsatisfiability by ensuring that every possible assignment is considered. The first complete approaches to solve the SAT problem are known as the DP [7] and DPLL [8] algorithms. Most modern SAT solvers are based on the principle methods established by the DPLL solver and are extended by many improvements.

This section gives a short overview of the concept of DPLL, while the subsequent section contains a more extensive review of the techniques used by modern SAT solvers.

The DPLL algorithm is defined recursively and takes only a formula Φ as input. It can be summarized into the following three main steps:

1. Check if Φ is satisfied or unsatisfied and return the result if either case occurs.

2. Check if any clause is unit and assign the variables implied by unit clauses.

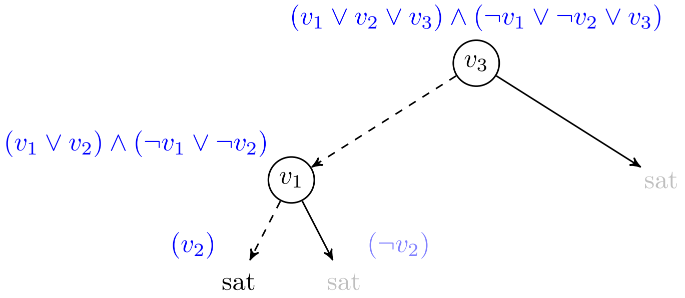
3. Choose a free variable v, called the decision variable. Check if $DPLL(\Phi|_{v \rightarrow false})$ or $DPLL(\Phi|_{v \rightarrow true})$ returns satisfiable. Otherwise return unsatisfiable.

A possible solve process for Formula 2.2 is shown in Figure 2.1. This graph is known as a decision tree because it visualizes the decision of the solver. The variable within the nodes is the chosen decision variable. The dashed arrow represents the assignment of this variable to $false$, the solid line that to $true$. The nodes that are reached by the arrows are the children of the origin node (which is also known as the parent node). The dashed line leads to the negative branch of the parent node, the solid line to the positive branch. Each node is annotated with the corresponding residual formula. The distance of a node to the tree root is its decision level.

In the example the solver first chooses v_3 as decision variable and assigns it to $false$. Next, v_1 is chosen and assigned to $false$. This leads to the residual formula (v_2) which is unit and forces the assignment of $v_2 = true$. At this point the formula is satisfied and the solver would return the model

$$\pi = \{v_1 \rightarrow false, v_2 \rightarrow true, v_3 \rightarrow false\}.$$

The remaining steps that are shown in light gray are only added for further visualization to highlight other possible outcomes, had the solver assigned some of the variables to $true$ first.



$(v_1 \vee v_2 \vee v_3) \wedge (\neg v_1 \vee \neg v_2 \vee v_3)$
v_3
$(v_1 \vee v_2) \wedge (\neg v_1 \vee \neg v_2)$
v_1
sat
(v_2)
$(\neg v_2)$
sat
sat

Figure 2.1.: Visualization of the DPLL solve process for Formula 2.2.

Modern SAT-Solving Techniques

The basic concept of a modern branch-and-bound SAT solver is still based on the DPLL algorithm. However, different changes and extensions have been introduced which lead to a conflict-driven clause-learning (CDCL) methodology. This section introduces the most sweeping changes that are implemented in virtually every successful SAT solver:

- Iterative computation: Unlike the classic DPLL algorithm, modern SAT solvers usually do not use recursion but a more efficient solve loop.

- Decision heuristic: The choice of the next variable that should be assigned is performed by a complex heuristic. Among the best known is the variable state independent decaying sum (VSIDS) heuristic [9] which measures how often a variable recently occurred in a conflict clause. The heuristic picks the free variable with the highest score for the next decision.

- Conflict learning: Whenever the solver encounters a conflict – to satisfy the formula the solver would have to assign a variable to $true$ and $false$ at the same time – it learns a new conflict clause. This clause attempts to store the essence of the conflict and ensures the solver will avoid this particular variable assignment in the future.

- Non-chronological backtracking: If the solver encounters a conflict it backtracks – removes the variable assignments and their implications – to the first decision level where the conflict clause becomes unit. This unit clause immediately implies a new variable assignment which should lead the solver away from the conflict.
- Restarts: If the solver cannot find a model after a certain amount of decisions it resets itself and starts a new solving attempt. Some of the learned knowledge is retained to ensure that the solver does not repeat the exact same calculations. The restart interval is increased over time.
- Preprocessing: By analyzing the formula it can often be drastically simplified. Such simplifications can, among others, find variables that must have a certain value or eliminate some variables completely.

Implementation details and many more improvements can be found in recent publications on SAT solvers [9]–[14] in addition to [1].

Incremental Solving

Modern SAT solvers often provide an incremental solving mode [10]. In this mode two additional features are supported. Firstly, after a formula has been solved it can be extended by adding new clauses and then solved again. Should the solver have learned any conflict clauses during the previous run, this knowledge is retained and used during any subsequent calls on the same base formula.

Secondly, incremental solvers support assumptions. An assumption forces a variable to a fixed value during the next solver call. However, the assumption can be reverted after the current formula was solved. The solver will ensure that it only retains knowledge that is independent from any assumed variable values. The combination of incremental additions to the base formula and assumptions allows for many advances, for example bounded model checking [15]. In this thesis, incremental solving is used to greatly increase the speed of the presented SAT-based algorithms.

2.1.4. Model Counting

Solving the Boolean satisfiability problem gives a single model for the formula (if one exists). Determining the number of different models of a formula is the closely related #SAT problem. From the number of satisfying assignments (called model count) it might be possible to derive the probability for the effect that is modeled by the formula to occur: When solving a formula with a SAT solver a single example for a solution is returned. However, from the model count – computed by a #SAT solver – the likelihood that the modeled problem is solved can often be computed. The #SAT problem is even more difficult to solve than the SAT problem: It is #P-complete [16].

A #SAT solver computes the model count of a given formula Φ. Similar to SAT solvers, two different approaches can be distinguished: Approximating #SAT solvers compute an approximation of the model count by using stochastic methods. This approximation is usually given as an interval with a certain confidence.

This work focuses on exact #SAT solvers which accurately compute the number of satisfying assignment by considering the entire decision tree much like a complete SAT solver does.

This thesis discusses the development of the #SAT solver `countAntom` and its distributed variant `dCountAntom` in detail in Chapter 5 and applications of #SAT to circuit testing in Chapter 6. This section will therefore only give an example for the general approach of exact #SAT-solving.

A #SAT solver works similar to the DPLL algorithm used for SAT-solving. However, instead of stopping when a satisfying assignment has been found, the solver continues with computing the model count for every branch of the decision tree until the entire decision tree has been traversed. The model count of the current branch is based on the number of remaining free variables: For n free variables there are 2^n different possible variable assignments in the current branch.

Figure 2.2 shows an example of the solving process for Formula 2.2. In comparison to the DPLL solving process in Figure 2.1, the #SAT solver definitely considers the entire decision tree and computes the model count for each node (shown below the corresponding nodes). The model count of an internal node within the tree is computed by adding the model counts of the two child nodes. The overall model count of Φ, abbreviated as $mc(\Phi)$, is 6.

Since there are $2^3 = 8$ possible different variable assignments, the satisfiability probability of Φ is $\frac{6}{8} = 75\,\%$.

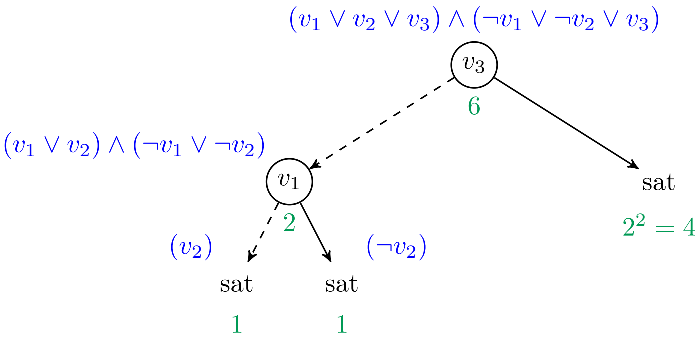

$(v_1 \vee v_2 \vee v_3) \wedge (\neg v_1 \vee \neg v_2 \vee v_3)$
v_3
6
$(v_1 \vee v_2) \wedge (\neg v_1 \vee \neg v_2)$
v_1
2
sat
$2^2 = 4$
(v_2)
$(\neg v_2)$
sat
sat
1
1

Figure 2.2.: Visualization of the #SAT solve process for Formula 2.2.

2.2. Circuit Testing

In this section the basics of circuit testing are introduced. Section 2.2.1 describes the general structure of digital combinational and sequential circuits. In Section 2.2.2 various approaches to improve the testability of a circuit are discussed. Section 2.2.3 gives a more in-depth insight into the real hardware implementations of logic cells. Next, Section 2.2.4 discusses the modeling of defects with fault models. Section 2.2.5 introduces the general problem of automatically generating test patterns for the modeled faults. In Section 2.2.6 SAT-based test pattern generation is described. Finally, Section 2.2.7 gives an overview of the techniques that are required to model the circuit's timing in a SAT-based test pattern generation approach.

2.2.1. Digital Circuits

A digital circuit consists of logic blocks that are connected by wires and implements a Boolean function $g : \mathcal{B}^n \rightarrow \mathcal{B}^m$ (where $\mathcal{B} \in \{0, 1\}$) with n inputs and m outputs. Usually, a circuit has controllable inputs (known as primary inputs) as well as observable outputs (known as primary outputs). The logic blocks in a circuit are known as cells and are defined in a cell library. These cells can be simple buffers or inverters, implement basic Boolean functions or incorporate more complex behavior (e.g., a combination of AND, OR and negation). The cell library describes the functionality of the cells using basic logic gates that can be directly mapped to Boolean operators. Thus, a circuit consisting of cells can be converted into a mapped circuit consisting of gates by replacing each cell by its definition. The mapped circuit can then be transformed into a Boolean formula. To distinguish the cell level description of a circuit from the description as a Boolean formula, '0' and '1' are used for the values of signals in a circuit while $false$ and $true$ are reserved for variables in a Boolean formula.

Tables 2.5 and 2.6 give an overview of the Boolean functions implemented by basic and complex cells, respectively. Many of the basic cells also exist in versions with more than two inputs (e.g., a four-input OR-cell). Instead of mapping these cells to multiple two-input gates, they are mapped to a single gate with the corresponding number of inputs (e.g., a four-input OR-gate). These gates can be transformed into CNF with less clauses than the multiple two-input gates could be: Directly encoding a four-input OR-cell requires five clauses. Mapping the cell to two-input cells requires nine clauses and two additional variables.

The presented cells are just examples. Generally, any Boolean function can be represented as a cell and used in a circuit. More details on the hardware implementation of the cells is given in the next section. In this work basic and complex cell are distinguished. A complex cell implements more advanced functions whereas a basic cell corresponds directly to a gate.

From a modeling perspective, a digital circuit can be seen as a directed graph with the logic blocks as nodes and the wires as edges. Figure 2.3 gives an example for a simple circuit consisting of three cells and a corresponding representation as a graph. In the graph, the nodes are annotated with the type of the cell they represent.

The representation as a graph is used for all of the algorithms presented in this thesis.

Table 2.5.: Basic cells and their corresponding Boolean functions.

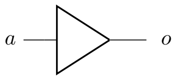
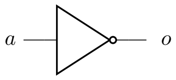
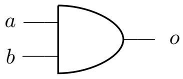
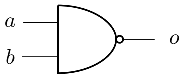
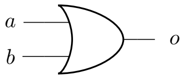
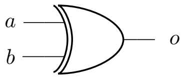
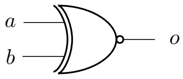
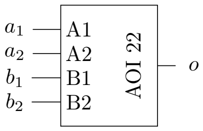
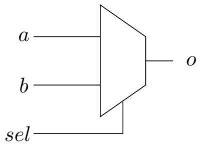
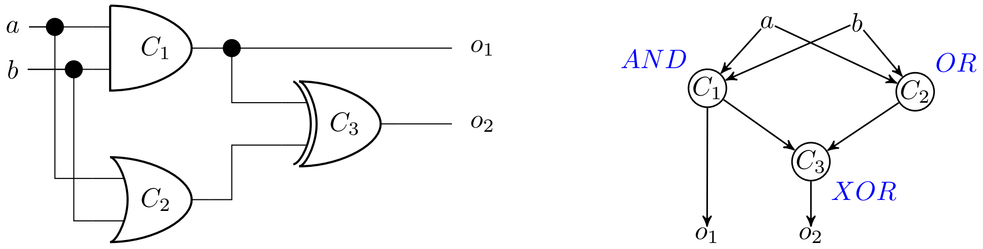
Name	Symbol	Boolean function
BUF		$o = a$
NOT or inverter		$o = \neg a$
AND		$o = a \wedge b$
NAND		$o = \neg(a \wedge b)$
OR		$o = a \vee b$
XOR		$o = a \oplus b$
XNOR		$o = \neg(a \oplus b) = (a \Leftrightarrow b)$

Table 2.6.: Two complex cells with their corresponding Boolean functions.

Name	Symbol	Boolean function
AND-OR-Invert (AOI)	 a_1 a_2 b_1 b_2 A1 A2 B1 B2 AOI 22 o	$o = \neg\big((a_1 \wedge a_2) \vee (b_1 \wedge b_2)\big)$
Multiplexer (MUX)		$o = \big((a \wedge sel) \vee (b \wedge \neg sel)\big)$

Thus, when discussing digital circuits in the context of algorithms, cells (or gates, when talking about a mapped circuit) are also referred to as nodes. However, the graphical representation as a circuit consisting of cells is chosen for clarity because it makes the overall structure of the circuit easier to comprehend.

The graph representing the circuit in Figure 2.3 is a directed acyclic graph: There is no cycle when following the edge only in the indicated direction. Generally, a circuit can also contain feedback loops which result in a cyclic graph. Circuits with feedback loops have a different and more complex behavior that is not always well defined in the purely digital domain (e.g., depending on analog characteristics of the manufactured circuit) and are, therefore, not considered in this thesis.



(a) As a circuit

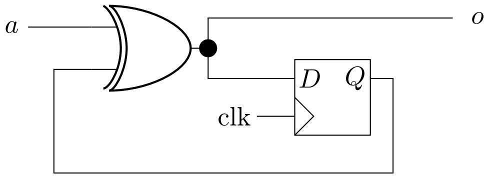
(b) As a directed graph

Figure 2.3.: Two different representations for the same digital circuit.

Memory Elements

In addition to cells, digital circuits can also contain flip-flops that act as memory elements. Flip-flops store the value on their input on a transition of a special signal, usually called clock. Circuits with memory elements are known as sequential circuits because the clock gives the circuit a defined sequential behavior, whereas circuits consisting only of cells are called combinational circuits. Figure 2.4 shows a small sequential circuit with one flip-flop and one cell. The flip-flop stores the value at its input D at a rising edge of the clock clk and shows it at the output Q. The output of the complete circuit is '1' when the value of a differs from that stored in the flip-flop.

There are many different types of storage elements in modern circuits and even many different kinds of flip-flops. For the context of this thesis, the handling of rising edge clocked D-type flip-flops like the one used in Figure 2.4 is presented. More details on different kinds of storage elements can, for example, be found in [17].



input a	flip-flop Q	output o
0	0	0
0	1	1
1	0	1
1	1	0

(a) The sequential circuit

(b) The implemented function as a table

Figure 2.4.: A small sequential circuit and the Boolean function it implements.

A sequential circuit can be split into two parts: the flip-flops and the remaining cells (called the combinational core). Figure 2.5 visualizes this partitioning.

Sequential circuits differ from circuits with a feedback loop because the value on the feedback lines only changes when there is a rising clock edge. Hence, for a single time frame starting after a rising clock edge until the next rising clock edge only the current output values of the flip-flops, the values at the primary inputs and the combinational core need to be considered. Therefore, the outputs of the flip-flops are considered to be secondary inputs of the combinational core. Similarly, the flip-flop inputs are considered as secondary outputs.

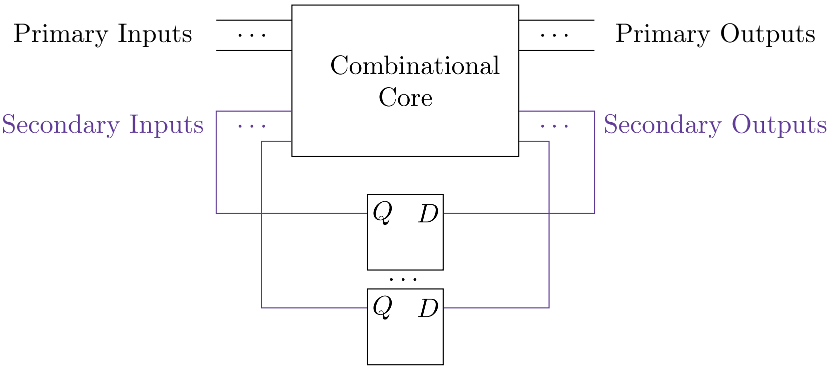


Figure 2.5.: Partitioning a sequential circuit into its combinational core and flip-flops.

2.2.2. Improving the Testability of a Circuit

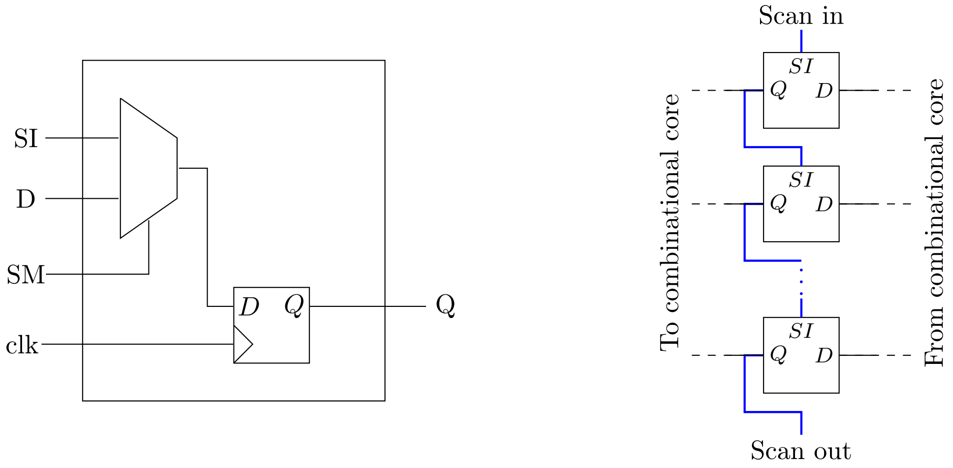
The testability of a fault depends on the structure and functionality of the corresponding circuit. Especially in sequential circuits it can be very difficult to generate the required input stimuli or to observe the fault effect. Imagine, for example, an n-bit counter that counts from 0 to $2^n - 1$ and a fault that requires the counter value to be “1. . . 1”. To reach this value, the counter would have to be incremented $2^n - 1$ times before a test could even be applied.

To solve observability and controllability challenges, various design for testability (DFT) improvements have been proposed [18].

Scan-Chains

One of the most common DFT techniques is the use of scan-chains to control the values of flip-flops in sequential circuits. A scan-chain connects flip-flops into a shift register that can be filled with values while the circuit is in test mode. Additionally, the values stored in the flip-flops can be shifted out of the chain again which makes them observable. To this end, flip-flops have to be augmented with additional logic to create a scan flip-flop (see Figure 2.6a). These scan flip-flops are then combined into a scan-chain (see Figure 2.6b). When the scan mode (SM) signal is ‘1’, the flip-flops act as a scan-chain and store the value at the scan-in (SI) port. When the scan mode is disabled, a scan flip-flop can be used just like any other flip-flop. In a full-scan circuit all of the flip-flops are accessible through scan-chains. This makes all of the secondary inputs controllable and all of the secondary outputs observable. Thus, the ATPG algorithms need only consider the combinational core of the circuit and do not need to handle the flip-flops. Faults in the additional scan logic are tested through separate and specialized methods [19]–[21] which are not pursued in this thesis.

In this work, it is assumed that all circuits are full-scan and that all secondary circuit inputs and outputs are, therefore, fully controllable and observable.



(a) Layout of a scan flip-flop based on a normal D-flip-flop.

(b) Structure of a scan-chain (in blue). The clock and test mode signals are not drawn.

Figure 2.6.: Construction of a scan-chain based on scan flip-flops.

Multiple Time Frame Tests

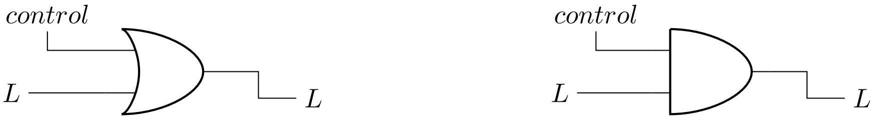
While scan-chains greatly improve the testability, shifting values into the chains still requires a certain amount of time depending on the chain length. Shifting in and out a complete pattern after every test leads not only to an increase in test time but can also cause issues with timing relevant tests. For example, in the transistor stuck-open fault model the detection of a fault depends on a floating line storing the charge from the first time frame. If the second test pattern T_2 takes a long time to be loaded into the flip-flops the charge might have already dissipated and the fault effect would be masked by the time T_2 is available.

To solve this problem, different techniques for multiple-pattern tests have been introduced: Launch-on-capture [22] (LOC) and launch-on-shift [23] (LOS) re-use (some of) the values that were used or generated in the previous time frame. In an LOS type test, the values in the flip-flops are shifted once more to derive the second pattern. For LOC tests the flip-flop values for the second pattern are derived from the secondary outputs of the previous time frame. Thus, the second test pattern can be applied only one clock-cycle after the first pattern but the values of the secondary inputs in all but the first time frame are limited by the possibilities of the combinational logic (LOC) or of the values in the scan-chain (LOS).

Other techniques, for example enhanced scan [24], allow for full freedom in all time frames but come at an increased cost and complexity.

Test Points

For some tests, it might be necessary to observe the value at a certain signal within the circuit or to set this signal to a specific value. This can be achieved by inserting a test point. To only observe a value, the signal can be routed to a pin of the device. To control the value of a signal, an additional cell needs to be inserted into the circuit. Figure 2.7 shows the required modifications to force a signal to '0' or to '1'. These two modifications can of course be combined to allow for an arbitrary signal value.



(a) When *control* is assigned to '1', L becomes '1', otherwise it is unchanged.

(b) When *control* is assigned to '0', L becomes '0', otherwise it is unchanged.

Figure 2.7.: Adding a test point to control the value of a line L in a circuit.

2.2.3. Cell Layout

Up until this point, digital circuits and the cells contained in them were considered only as Boolean functions. However, to build actual working hardware circuits, this abstract description has to be mapped to a physical implementation. Most modern logic is implemented as complementary metal-oxide-semiconductor (CMOS) devices [25]. MOS transistors have three ports: gate, source and drain. The gate input controls whether the drain and source ports are connected or not. The transistor is considered to be turned on when it is conducting. In CMOS, two types of transistors are used:

- N-type transistors: These transistors conduct when a voltage is applied to the gate.
- P-type transistors: These transistors conduct when no voltage is applied to the gate.

By combining these transistors into small circuits, the Boolean functions of cells can be implemented. Figure 2.8 shows the transistor layout of an OR-cell with inputs $A1$ and $A2$ and output ZN as implemented by the 45 nm Nangate library [26]. A cell library provides a set of different logic cells, their Boolean function as a gate description, physical implementation and often a characterization of the power and timing characteristics of the cells.

With the information provided by the cell library, a digital circuit can be transferred into a physical design, an integrated circuit (IC).

For the mapping between physical implementation, Boolean function and Boolean formula, the voltage levels of the circuit have to be interpreted. To this end, a logic '1' (or *true* in the context of Boolean formulas) represents the positive supply voltage V_{DD} of the circuit. Conversely, a logic '0' ($false$) is represented by the negative supply voltage V_{SS} – often this is also the ground level.

An example for the functionality of a CMOS logic cell is given with the transistor-stuck-open fault model discussed in the next section.

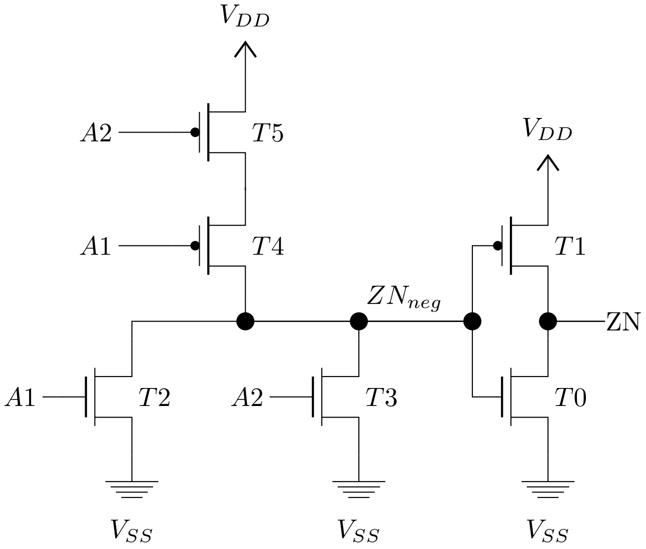


Figure 2.8.: Transistor layout of an OR-cell as implemented by [26].

2.2.4. Fault Models

There are many reasons why a manufactured IC does not function as intended, ranging from design bugs over incorrect specifications to manufacturing challenges. The field of circuit test focuses on finding manufacturing defects that influence the functionality of the produced chip. With the mass production of chips which have smaller and smaller feature sizes and more and more transistors, such defects are impossible to eradicate and are occurring with a certain probability for any manufactured chip. The percentage of chips without a defect is the yield of the production.

With an almost infinite number of possible defects, an abstraction is needed. In structural circuit test, a fault model is used to generate a finite list of faults that are in some way related to real defects. By testing a manufactured chip for the presence of any of the modeled faults, one hopes to separate the defect-free chips from those which are defective.

The quality of a fault model impacts the share of chips that pass the testing but still contain a defect, the so called test escapes. The number of defective parts that are shipped, usually counted in defective parts per million (DPPM), directly corresponds to the defective chips that escape all tests. While a low DPPM rate is definitely desirable (and often certain DPPM rates are required by a customer), the test cost is always a consideration, especially with cheap, mass manufactured devices. Thus, a balance between optimal fault or defect coverage and test application time has to be found.

Many different fault models have been proposed. This section discusses the well established stuck-at [27] and transition-delay [27], [28] fault models as well as the more advanced transistor stuck-open model [29]. A comprehensive collection of different models used in hardware testing can, for example, be found in [2].

To test a circuit for the presence of a modeled fault, a test pattern is applied to the inputs of the circuit. This test pattern is a specifically crafted input assignment with the property of creating a difference between the fault-free and the fault-affected circuit on at least one of the outputs. A test pattern fulfills two tasks: Firstly, it activates the fault which makes it visible at the faulty location. This step usually involves applying a detection pattern to the fault site. Secondly, it propagates the fault effect to at least one of the observable circuit outputs. The signal values within the fault-free version of the circuit are called the good (G) values, those in the fault-affected circuit the bad (B) values.

For sequential circuits, extra care has to be taken to ensure a correct handling of the secondary inputs and outputs because these are actually connected to flip-flops and not directly controllable and observable. This special handling is discussed in Section 2.2.2.

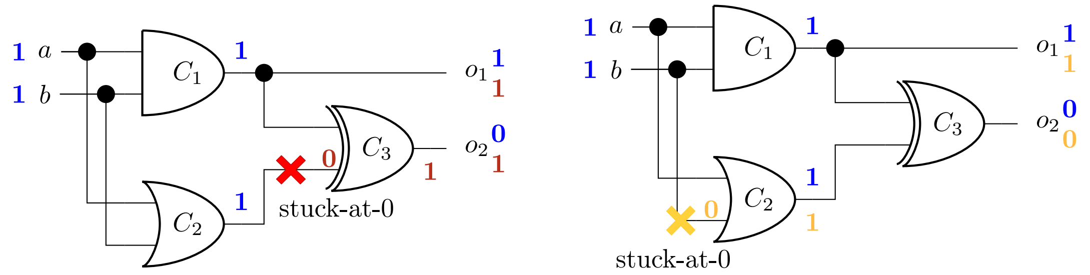
Stuck-At Faults

In the single stuck-at fault model one of the inputs or the output of one of the cells in the circuit is always '1' or always '0'. Thus, the number of different faults is limited to twice the sum of the number of cell inputs and outputs within the circuit. Furthermore, many stuck-at faults are equivalent and can be combined. Consider, for example, a stuck-at-1 fault at the input of an inverter and a stuck-at-0 fault at its output. The stuck-at-1 fault would always produce a '0' at the output of the inverter which is equivalent to a stuck-at-0 fault at the output. Hence, it is sufficient to test for the absence of one of the two faults.

An example for stuck-at faults is shown in the circuit in Figure 2.9. Here, two different stuck-at faults and the effect of the test pattern "11" are analyzed. The propagation of the test pattern in the fault-free circuit is shown in blue. The yellow and red values show the values in the faulty circuits starting from the particular fault location.

The pattern "11" is a test pattern for the stuck-at-0 fault at the second input of C_3 (Figure 2.9a). It activates the fault location by applying the good value '1' to the input which differs from the bad value '0'. The fault effect is then propagated through the XOR-cell to output o_2 where a difference between the good and bad output value can be observed.

On the other hand, the pattern "11" is not a test pattern for the stuck-at-0 fault at the second input of C_2 (Figure 2.9b). While the fault site is successfully activated, the fault effect is not propagated through the OR-cell C_2. Hence, there is also no difference at either output of the circuit.



(a) Stuck-at-0 fault at the second input of C_3 (b) Stuck-at-0 fault at the second input of C_2

Figure 2.9.: Two circuits with different stuck-at faults.

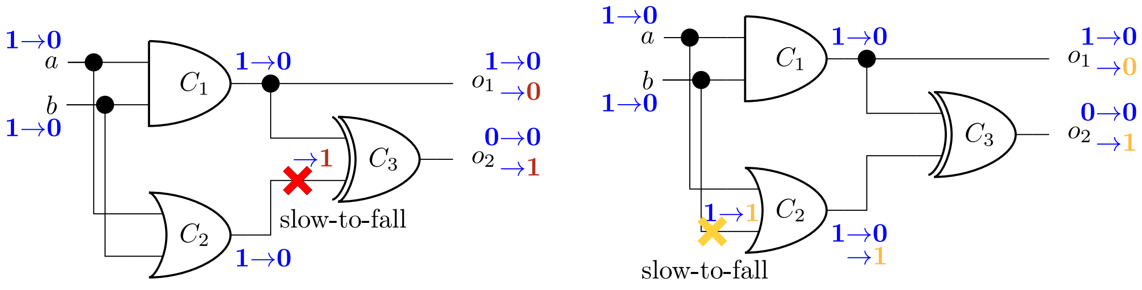
Transition-Delay Faults

The stuck-at fault model assumes that the faulty signal is stuck at a specific value. While this covers many different defects and even though the stuck-at model is widely used throughout the industry, testing only for stuck-at faults is not sufficient to ensure the absence of all possible defects. The single transition-delay fault model targets defects which do not cause a line to be stuck but instead delay the propagation of a rising or falling transition for a long time. Thus, when the assignments T_1 and T_2 are successively applied to the circuit's inputs, the faulty line will remain at the value implied by T_1 and will not change to the value implied by T_2. The transition delay fault locations are chosen similarly to the stuck-at model. In addition, the delay faults are further refined by distinguishing slow-to-rise and slow-to-fall faults.

It should be noted that the transition-delay model assumes that the propagation of a transition is delayed for a long time (until after the next clock edge triggers the flip-flops to be updated). However, the propagation of the transition still eventually occurs. Other fault models, for example the small delay fault model, assume that the signal propagation is only delayed for a certain (small) amount of time [30]–[32] and can be used to target defects which only cause slight changes in the circuit's timing.

To test for a transition delay fault, a two pattern test $\langle T_1, T_2 \rangle$ has to be applied to the circuit. The initialization pattern T_1 generates a known logic value at every line in the circuit and the required logic value at the fault site (i.e., a '0' for a slow-to-rise fault and a '1' for a slow-to-fall fault). In the second time frame, the propagation pattern T_2 activates the fault and propagates its effect to at least one output. T_2 is also known as the launch pattern since it launches the transition that tests for the transition-delay fault.

Figure 2.10 shows an example for the detection of two different transition-delay faults. The test pattern $\langle 11, 00 \rangle$ is able to detect both of the slow-to-fall faults at output o_2 because the output has a value of '1' in the fault-affected circuit instead of the correct '0'. In both cases a $1 \to 0$ transition is delayed and the corresponding signal stays at '1'.



(a) Slow-to-fall fault at the second input of C_3 (b) Slow-to-fall fault at the second input of C_2

Figure 2.10.: Two circuits with different transition-delay faults.

Transistor Stuck-Open Faults

The transistor stuck-open fault (TSOF) model represents a specific kind of fault: opens. Recent data from manufactured devices shows that opens are a predominant defect mechanism in modern devices [33]–[35]. An open can occur within a cell (intra-cell open) or on the interconnects between cells and be partial, resistive or full.

The TSOF model assumes that a single transistor within a cell will never conduct and is always open. This corresponds to defects within the transistor itself or within the wires that connect the terminals of the transistor to its neighbors. This fault model is also known as the transistor stuck-off model which describes the exact same effect.

Some TSOFs are likely to be covered by sets of stuck-at or transition-delay tests. However, others would most likely be missed without specifically designed tests. This thesis presents novel approaches for the generation of test patterns for TSOFs which accurately account for many of the challenges that arise with a detailed low-level fault model where timing and analog effects cannot be ignored anymore. Therefore, this section will only give some examples for the test of TSOFs while a more exact discussion of the fault model is deferred to Chapter 4.

Figure 2.11 shows the effect of a TSOF at the transistor level of an OR-cell. Similar to transition-delay faults a two pattern test is required for the detection of these faults. Here, the transistor $M2$ is stuck-open and the detection pattern $\langle 00, 10\rangle$ is applied to the cell inputs.

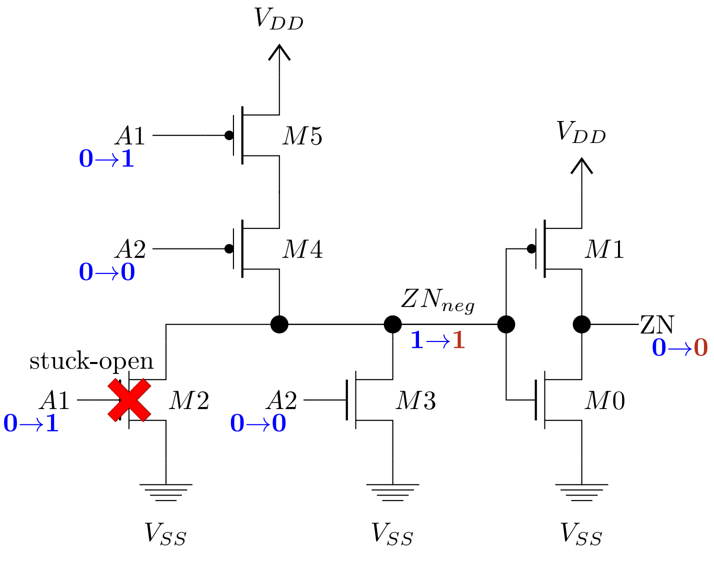


Figure 2.11.: The transistor layout of an OR-cell with a TSOF in transistor $M2$.

In the first time frame the initialization pattern "00" turns on $M4$ and $M5$. This creates a conducting path from V_{DD} to the internal wire ZN_{neg} which is charged to '1'. This turns on $M0$ and creates a conducting path between V_{SS} and ZN. Thus, the cell output becomes '0'. So far, the cell behaves normally: the faulty transistor $M2$ is off anyway, since $A1$ is '0'.

In the second time frame, the propagation pattern "10" turns off $M5$ and should turn on $M2$. However, as $M2$ is stuck-open, ZN_{neg} is connected to neither V_{DD} nor to V_{SS}. Instead, it is floating and maintaining its previous charge and value due to the line capacitances [29]. Therefore, the circuit output ZN stays at '0' – the correct output value would be '1'.

The TSOF in transistor $M2$ of the considered OR-cell is not represented by any stuck-at or transition delay-fault. It is, therefore, not reliably detected by a stuck-at or transition-delay test pattern:

- **Stuck-at:** The cell output ZN can become both '0' or '1' because $M3$ offers a possible parallel path to the faulty $M2$. Thus, the fault is not a stuck-at fault and no stuck-at pattern could guarantee a successful detection. The pattern "10" is a detection pattern for a stuck-at-0 fault at the cell. However, without knowing (or controlling) the charge of ZN_{neg}, the output might very well already be '1' and the cell would be considered fault-free.

- **Transition-Delay:** The pattern $\langle 00, 10\rangle$ (which detects the TSOF in $M2$) is also a detection pattern for a slow-to-rise fault at the output of the OR-cell. However, so are $\langle 00, 01\rangle$ and $\langle 00, 11\rangle$ both of which fail to detect the TSOF in $M2$ (because they open $M3$). Therefore, when picking a random transition delay pattern, the detection probability for the TSOF would be only $\frac{1}{3}$.

As a result, a specifically crafted TSOF test pattern is required to guarantee a successful test of the fault and the defects which might be causing it.

2.2.5. Automatic Test Pattern Generation

An automatic test pattern generation (ATPG) algorithm computes a test pattern for a fault in a given circuit. Usually, an ATPG algorithm is used to generate a test pattern for every single fault in the circuit. These test patterns can then be applied to a manufactured IC. Depending on the circuit structure and the fault model, there are usually some faults for which a test pattern cannot be generated. The share of faults which is detected by at least one test pattern is known as fault coverage. Depending on the fault model, a high fault coverage also gives a high defect coverage, which in turn should result in a low DPPM rate.

Many different ATPG algorithms have been proposed. They range from the D-algorithm [36] and its variants [37], [38] which work more or less directly on the circuit structure, to advanced SAT-based techniques which utilize an abstract model of the test pattern generation problem and are applicable to a wide range of scenarios [39]–[48].

The exact nature of the ATPG algorithm depends on the considered fault model. The general idea, however, is always the same: The ATPG algorithm must compute a test pattern that creates a difference on at least one observable output if the considered fault is present in the circuit.

2.2.6. SAT-Based Test Pattern Generation

This thesis focuses on SAT-based ATPG algorithms. Here, the problem of generating a test pattern is transferred to a Boolean formula which can be solved by a SAT solver. If the SAT solver finds a model for the formula, a test pattern can be derived from the variable assignment.

The basic concept of SAT-based ATPG for a fault f can be divided into three steps each of which is described in detail in the subsequent sections:

1. Create a miter circuit [49] with a good and bad copy of the circuit.
2. Create a representation of the miter circuit as a Boolean formula Φ.
3. Add the fault activation condition to Φ.

By solving Φ with a SAT solver a test pattern for f is computed. If Φ is unsatisfiable, it is proven that the fault is untestable.

Creating the Miter Circuit

Figure 2.12 shows the general structure of a miter circuit for ATPG. It consists of two copies of the original circuit which are used to compute the output values for the fault-free (good) and fault-affected (bad) circuit. Corresponding inputs of the circuit copies are connected to form a new input of the miter circuit. This ensures that the inputs of the good and bad circuit have the same value. To detect a fault, the value of at least one of the outputs of the bad circuit must differ from that of its representative in the good circuit. To detect this difference, an XOR-cell, which combines corresponding outputs, is added for each pair of outputs. Finally, the outputs of the difference-detecting XOR-cells are connected by an OR-cell. If the output of the OR-cell is '1', at least one of the outputs shows a difference.

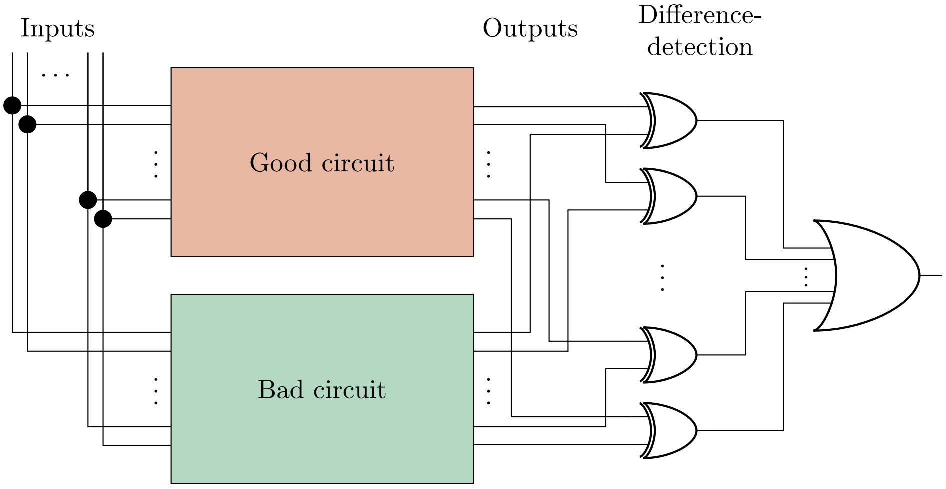


Figure 2.12.: The layout of a miter circuit for test pattern generation.

Converting the Miter Circuit to a Boolean Formula

The miter circuit has to be converted into a format that can be handled by a SAT solver – a Boolean formula. Luckily, the circuit can easily be transformed into a mapped circuit consisting of logic gates. The logic gates are closely related to Boolean formulas already since they correspond to Boolean operators. In fact, the entire mapped circuit could be converted into a Boolean formula by traversing the circuit from the inputs to the output and combining the encountered Boolean functions of the gates into an ever larger formula.

As an example consider the mapped circuit in Figure 2.13a. Here, every gate is annotated with the (sub-)formula that is contained until this stage of the circuit. Therefore, the Boolean function implemented by the circuit is represented by the following Boolean formula:

$$\Phi = \Big(o \Leftrightarrow \big((a \wedge b) \oplus (a \vee b)\big)\Big)$$

This direct conversion of the circuit is not in CNF and Φ can, therefore, not be directly used by most SAT solvers. The formula has to be transformed into CNF with the Tseitin transformation [4] which generates an equisatisfiable CNF that is linear in the size of the original formula.

Instead of first creating a huge, convoluted formula and then transforming this formula to CNF, one can also apply the Tseitin transformation to the mapped circuit on a gate-by-gate basis. This direct conversion is shown in Figure 2.13b. The final formula Φ_{CNF} is simply the combination of the individual sub-formulas for each gate:

$$\begin{aligned}\Phi_{CNF} =& (a \vee \neg x) \wedge (b \vee \neg x) \wedge (\neg a \vee \neg b \vee x) \wedge (\neg a \vee x) \wedge (\neg b \vee x) \wedge (a \vee b \vee \neg x)\\ & \wedge (x \vee y \vee \neg o) \wedge (\neg x \vee \neg y \vee \neg o) \wedge (\neg x \vee y \vee o) \wedge (x \vee \neg y \vee o)\end{aligned}$$

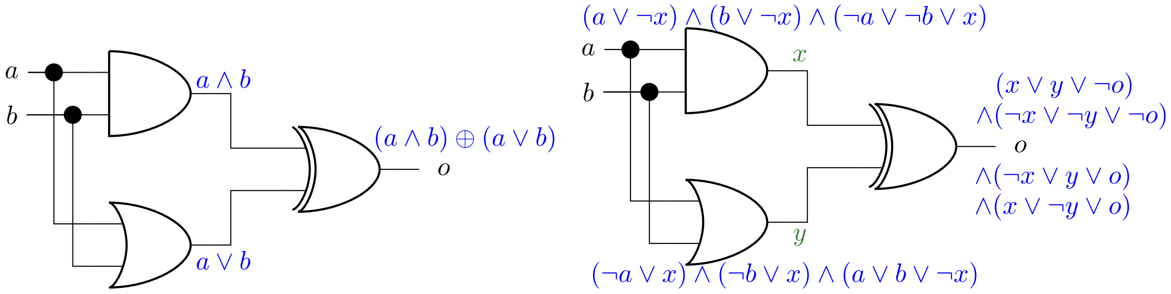
Φ_{CNF} directly encodes the behavior of the circuit in a SAT solver friendly format and can be easily generated by traversing the circuit once from the inputs to the outputs. Every variable in Φ corresponds to a signal in the miter circuit. After solving the formula, the value of the signal can simply be derived by the value of the variable in the model.

To increase readability and clarity, many Boolean formulas shown in this thesis are not in CNF and include implications and other boolean operators instead. These formulas are all transformed into CNF with the Tseitin transformation before being handed over to the SAT solver.

The Fault Activation Condition

The fault activation condition – that is, the values that need to be applied to the cell inputs to make the fault effect visible at its output – differs depending on the fault model. Thus, the modeling of the activation condition in the ATPG also differs and might also be implementation specific. The approach that is described in this section is the one that is implemented by the SAT-based ATPG algorithms developed for this thesis.

For stuck-at faults the faulty line is split into two parts. The first part is connected to the cell driving the line and must have a value that is inverse to the fault (i.e., ‘1’ for a stuck-at-0 fault and ‘0’ for a stuck-at-1 fault) to activate the fault. The second part is connected to all the cells driven by the line. This line has the value of the stuck-at fault in

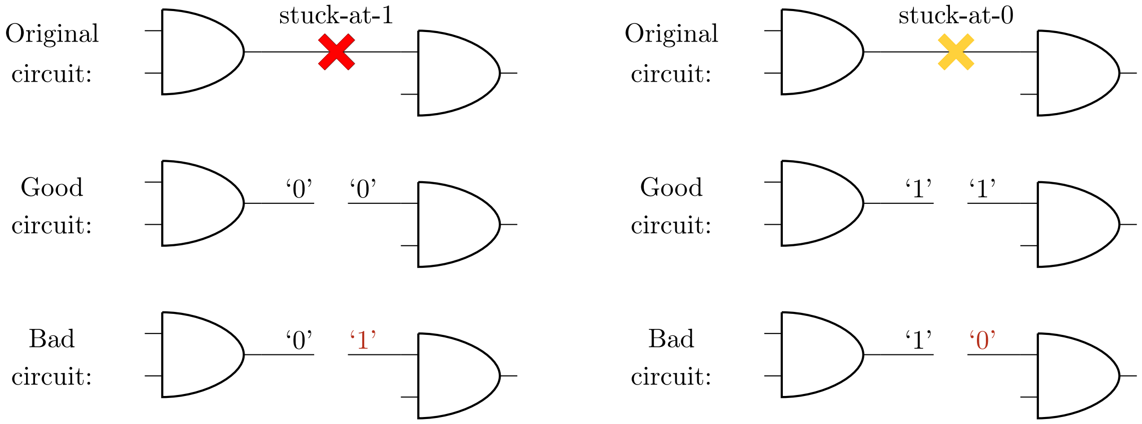

a
b
$a \wedge b$
$a \vee b$
$(a \wedge b) \oplus (a \vee b)$
o
$(a \vee \neg x) \wedge (b \vee \neg x) \wedge (\neg a \vee \neg b \vee x)$
a
b
x
$(x \vee y \vee \neg o)$
$\wedge(\neg x \vee \neg y \vee \neg o)$
o
$\wedge(\neg x \vee y \vee o)$
$\wedge(x \vee \neg y \vee o)$
y
$(\neg a \vee x) \wedge (\neg b \vee x) \wedge (a \vee b \vee \neg x)$

(a) Direct conversion with a gradual formula construction

(b) Tseitin transformation, which introduces additional helper variables

Figure 2.13.: Construction of the Boolean formula that represents the circuit.

the bad circuit and its inverse in the good circuit. Thus, the only difference between the good and the bad circuit lies with the values at the fault location. Figure 2.14 shows an example for the fault activation conditions for both types of stuck-at faults. The required values for the line are added as unit clauses to the Boolean formula.



(a) For a stuck-at-1 fault

(b) For a stuck-at-0 fault

Figure 2.14.: Modeling the fault activation condition and effect by splitting the faulty line.

This process shows one of the strengths of SAT-based ATPG: There are almost no limits with regard to the fault model, fault location and fault effect. The basic circuit modeling can remain unchanged and only the parts that are changed by the fault need to be added.

To generate a formula that creates a valid test pattern one more step is needed: It has to be ensured that at least one of the circuit outputs shows a difference between the fault-free and faulty versions of the circuit. In the miter circuit this is achieved by combining all of the difference-detecting XOR-cells through a large OR-cell and then forcing the output of this cell to be '1'. In SAT-based ATPG the OR-cell is not needed. Instead, the variables representing the outputs of the XOR-cells are combined into a single new clause. To satisfy the formula, this clause has to be satisfied. Thus, at least one of the XOR-output's variable has to be *true*.

2.2.7. Modeling Time in SAT-Based ATPG

The SAT-based ATPG approach that has been outlined in the previous sections can be extended to handle the concept of time. In the context of an ATPG algorithm, this thesis uses two different conceptions of time:

- A time frame: Here, time is considered to be progressing stepwise, for example, with every rising clock edge. This concept is very popular as it is used in common multiple time frame fault models like the transition-delay fault model.
- Waveform accurate time: In this conception, time is considered at a much finer granularity at which the propagation of a signal through the different cells can be analyzed with waveform accuracy. The different cell delays are extracted from the circuit layout specifications which gives a close representation of the real circuit's behavior.

This section describes how both of these conceptions of time can be modeled in a SAT-based ATPG algorithm.

Time Frames

To model the circuit's behavior across multiple time frames it is unrolled: For every considered time frame a copy of the circuit is added. Then, depending on the selected test type (LOS, LOC, . . .), the secondary inputs and outputs are connected. Figure 2.15 gives an example for a LOC type test over two time frames. Note that the primary inputs and outputs are always controllable and observable whereas only the secondary inputs of the first time frame can be controlled and only the secondary outputs of the second time frame can be observed.

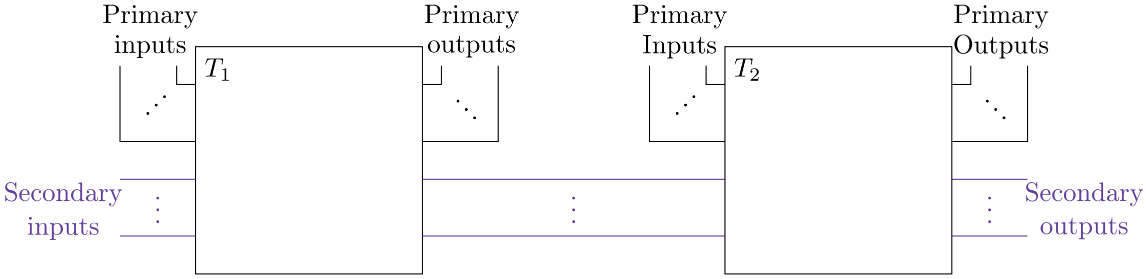


Figure 2.15.: Modeling of two time frames by unrolling the circuit.

The unrolled circuit can be used in a miter circuit like any other circuit. The only difference is that for the bad circuit the fault has to be added in each copy for every time frame.

Waveform Accurate Time

A waveform accurate SAT model of the transitions caused by a change in the input assignment was first presented in [48]. The approach can be summarized as follows: The continuous time is split into small discrete time steps. The resolution of the discretization can be chosen arbitrarily. Next, the earliest and latest possible time steps at which a signal can change its value are computed for every line in the circuit. This only requires a single sweep of the circuit from the inputs to the outputs. Finally, a copy of each cell is added for every time step where the cell might be active. The cell copy models the behavior at exactly one time step. Every signal is represented by a list $Tvars$ of variables representing the value of the signal from the earliest to the latest point of activity.

Figure 2.16 gives an example of the signal propagation of the pattern $\langle 11, 00 \rangle$ applied to the circuit from Figure 2.3a. The values after the @ symbol give the time step after which a transition occurs.

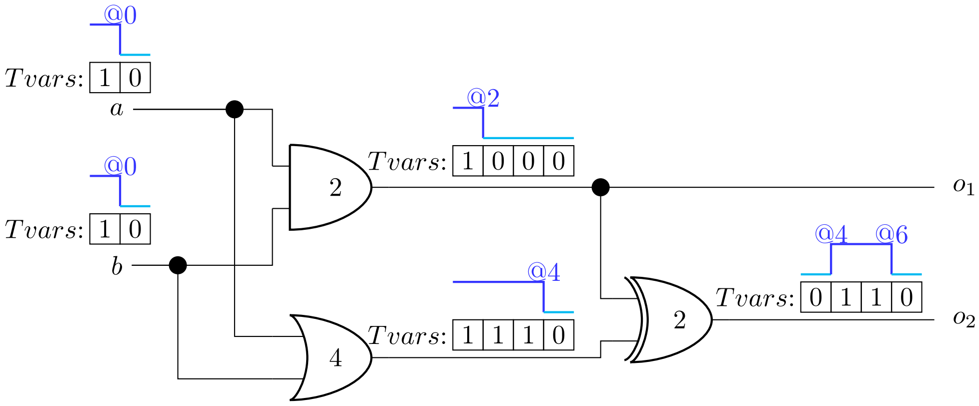


Figure 2.16.: Propagation of signal changes through a circuit. The value in each cell gives the delay from its inputs to its output.

This example also shows the relevance of a waveform accurate modeling: Due to the different delays along the paths to the XOR-cell, the falling transitions arrive at different time points. This causes a glitch at the output of the cell – an intermediate value on a signal that occurs only while the circuit is still propagating the changes which occur due to a change in input values. Glitches can have positive and negative effects on the testability of a fault. This topic will be discussed further in Chapter 4.

3. Advanced Modeling Techniques for SAT-Based ATPG

Since its introduction roughly 25 years ago [39], SAT-based ATPG algorithms have been broadly studied and applied to many different fault models and related problems. Such algorithms offer a remarkable performance on large industrial designs [41], [47]. Especially for untestable faults there appears to be a clear benefit to using SAT instead of working directly on the circuit structure [40].

Furthermore, the more abstract nature of the problem description makes it easy to consider higher value logics [45], [46], [50] or more sophisticated fault models [44], [51]–[53].

At the same time, the abstraction from a circuit to a Boolean formula can also remove some of the useful information. Introducing some of this information back into the Boolean formula can increase the overall solve speed by guiding the SAT solver through the search space more efficiently. This chapter discusses one such improvement: D-chains. A D-chain adds information about the propagation of the difference between the good and bad circuit to the Boolean formula used by the SAT-based ATPG algorithm. The concept itself was first introduced in [39] and further divided into a forward [39], [54] and backward [55] implication D-chain. In [56] a precursor of the presented good-diff D-chain was introduced. The indirect gate encoding presented in that work is, however, limited to gates with two inputs. Furthermore, a basic idea for a reduction of the redundancy created by D-chains was shortly discussed in [57] but not followed up any further.

This chapter discusses and compares the known D-chain variants and introduces the good-diff D-chain encoding which supports cells with any number of inputs as well as novel efficient hybrid implementations.

In Section 3.1 the implementation of an optimized SAT-based ATPG algorithm for the evaluation of the different D-chains is presented. Afterwards, the backward and forward D-chains are discussed in Section 3.2. The newly developed good-diff D-chain and its hybrid variants are introduced in Sections 3.3 and 3.4, respectively. In Section 3.5 a comprehensive analysis and comparison of the different D-chains across a large selection of different circuits is performed. Section 3.6 concludes the chapter with a summary of the contributions.

This chapter is partially based on:

[J2] P. Raiola, J. Burchard, F. Neubauer, D. Erb, and B. Becker, "Evaluating the effectiveness of D-chains in SAT-based ATPG and diagnostic TPG", *Journal of Electronic Testing: Theory and Applications (JETTA)*, 2017. DOI: `10.1007/s10836-017-5693-6`

[C1] J. Burchard, F. Neubauer, P. Raiola, D. Erb, and B. Becker, "Evaluating the effectiveness of D-chains in SAT-based ATPG", in *18th IEEE Latin American Test Symposium (LATS)*, 2017. DOI: `10.1109/LATW.2017.7906752`

The main contributions by the author to this chapter are:

- The implementation of a reference SAT-based stuck-at ATPG algorithm.
- The addition of different improvements to the ATPG to increase the overall solve speed.
- The implementation of a forward and a backward D-chain.
- The development and implementation of the good-diff D-chain.
- The development and implementation of two hybrid D-chain variants.
- The first comprehensive analysis of all of the different D-chains for a wide range of different circuits.

The implementations are built on top of the `PHAETON` framework [44] by Matthias Sauer which, among others, provides a circuit file parser, logic generators for the Tseitin transformation as well as an interface to the SAT solver `antom` [13], [14]. The formal generalization of the good-diff D-chain for multiple input cells and its proof of correctness was performed by Pascal Raiola.

3.1. Optimized Stuck-At ATPG

To evaluate algorithmic improvements, a baseline ATPG implementation is required. To this end, an optimized SAT-based stuck-at ATPG which combines support for complex cells and incremental solving is designed and implemented.

The basic concept of a SAT-based ATPG was already introduced in Section 2.2.6. In this section, the more advanced features that are required for an efficient test pattern generation are discussed. The three main improvements are:

1. Merely model the required parts of the circuit.
2. Accurately model complex cells.
3. Incrementally build and solve the formula.

Each of these improvements will be discussed in the following sections. By combining them, the presented algorithm is able to efficiently compute a test pattern for a given stuck-at fault. It should be noted that the discussed improvements are not novel and used by other SAT-based ATPG algorithms as well.

3.1.1. Modeling Merely the Required Parts of the Circuit

Not all parts of a circuit are needed to compute a test pattern for a fault [58]. Figure 3.1 shows the modeled parts for the stuck-at ATPG.

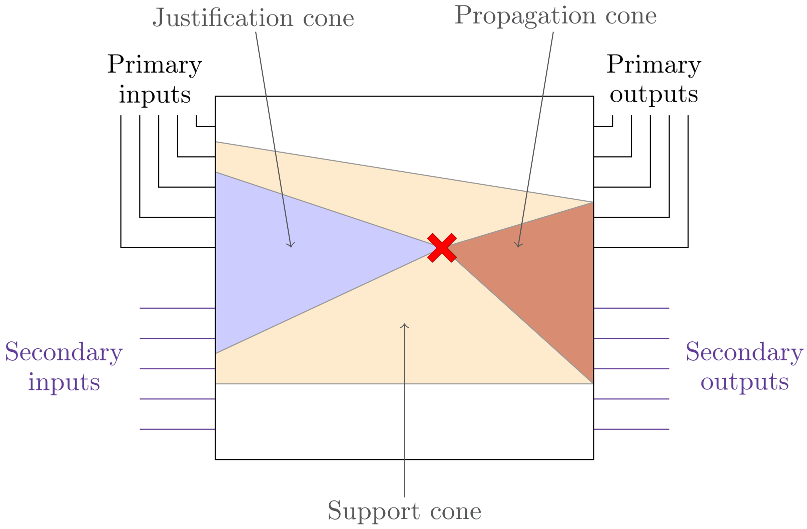


Figure 3.1.: Only the marked areas of the circuit are required when modeling a fault (red cross).

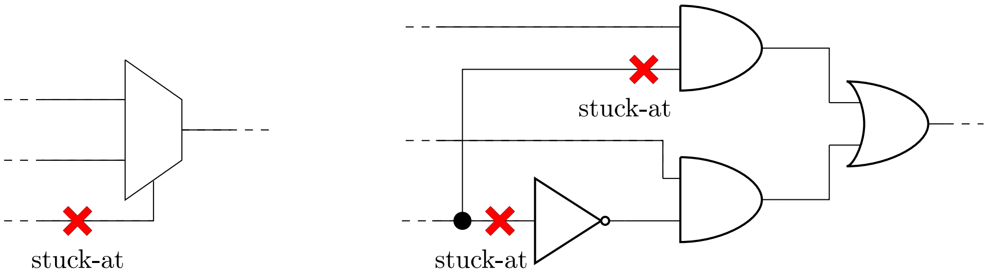
The effects of a fault in a circuit only propagate to the outputs that are structurally connected to the fault site. The cells that are connected to the output of the faulty cell are known as the propagation cone (shown in red). Similarly, only the inputs with a path leading to the fault site can potentially influence its activation. The corresponding cells form the justification cone (blue). Furthermore, the values of the side inputs of the cells in the propagation cone are required – these are provided by the support cone, marked in yellow.

By modeling only the required parts of the circuit, the size of the formula can be drastically reduced.

3.1.2. Accurately Modeling Complex Cells

Modern cell libraries and circuits rely heavily on complex cells because they allow for a tighter integration of logic functions, resulting in a smaller footprint and lower delays. In the PHAETON framework the original circuit is converted into a mapped circuit by replacing every complex cell by its definition based on logic gates. This creates a mapped circuit which only consists of gates. These gates can then be transformed into a formula with the Tseitin transformation and the ATPG can run as previously described in Chapter 2.

When modeling a stuck-at fault in the original circuit, the fault also has to be transferred into the mapped circuit. Since the input signal of a cell might be connected to more than one gate in the cell's definition, a stuck-at fault at a cell input might correspond to multiple stuck-at faults in the mapped circuit (see Figure 3.2). To appropriately handle these effects, the ATPG algorithm is extended to support multiple different stuck-at faults in the mapped circuit. This ensures that all kinds of complex cells are accurately represented in the ATPG flow.



(a) A multiplexer with a stuck-at fault at the select input.

(b) The logic gate definition of the multiplexer requires two stuck-at faults.

Figure 3.2.: A single stuck-at fault at the input of a complex cell might correspond to multiple stuck-at faults in the mapped circuit.

3.1.3. Incremental Solving

The normal SAT-based ATPG approach is based on a miter circuit with an XOR-cell between all corresponding outputs that could possibly show a difference. For an incremental solving approach [47], the miter is instead built up step-by-step to create smaller formulas that can potentially be solved more quickly.

Let O_f be the list of outputs in the propagation cone of the fault. In the incremental approach, in the first step, a single output $o_1 \in O_f$ is chosen and the initial formula is created. This formula models the entire justification cone of the fault but only the propagation to o_1. This results in a potentially much smaller propagation and support cone. The formula is then solved. If it is satisfiable, a test pattern that makes the fault visible at o_1 has been found – and the ATPG for this fault is finished. Otherwise, it is proven that the fault effect cannot be propagated to o_1. In this case, the next output $o_2 \in O_f$ is selected and the formula augmented with the missing information required for the propagation to o_2 and the necessary support. Because it is already known that the fault cannot be propagated to o_1, the output of the XOR-cell that detects a difference at o_1 is forced to '0'. This ensures that the solver does not perform any unnecessary calculations. The new formula is then solved again. From here on, the algorithm is repeated until a test pattern has been found, or it has been proven for all different outputs in O_f that the fault cannot be propagated there.

The algorithm is summarized in Figure 3.3. For the solving itself, the SAT solver is used incrementally. This allows it to maintain helpful knowledge of the previous solver call and further improves the solve speed.

```
$o_1 = O_f$.pop();
$\Phi$ = encodeMiter($o_1$) ;
solveResult = solve($\Phi$ + assumeDifferenceAt($o_1$));
if solveResult.isSAT() then
    return extractTestPattern(solveResult);
end
$o_{last} = o_1$;
while not $O_f$.empty() do
    $o_i = O_f$.pop();
    $\Phi$ += encodePropagationCone($o_i$);
    $\Phi$ += encodeSupportCone($o_i$);
    $\Phi$ += encodeNoDifferenceAt($o_{last}$);
    solveResult = solve($\Phi$ + assumeDifferenceAt($o_i$));
    if solveResult.isSAT() then
        return extractTestPattern(solveResult);
    end
    $o_{last} = o_i$;
end
return "untestable";
```

Figure 3.3.: Incremental SAT-based ATPG framework with O_f implemented as a stack.

3.2. D-Chains

A D-chain is an optimization technique in a SAT-based ATPG algorithm that enhances the generated formula with information regarding the propagation of the differences between the good and bad circuit. It should be noted that the D-chain is defined on the mapped circuit. Therefore, for the following definitions the circuit is assumed to be mapped to logic gates already.

Consider the circuit in Figure 3.4. Each line is annotated with the information that a classic SAT-based ATPG provides: variables representing the good (G) and bad (B) circuit and extra variables for the difference at each output (Do).

Note that the gate C_2 is not in the propagation cone of the fault and does not require a B variable. Since it is in the support cone of C_4 (which is located in the fault propagation cone), C_2 still has to be modeled in the good circuit.

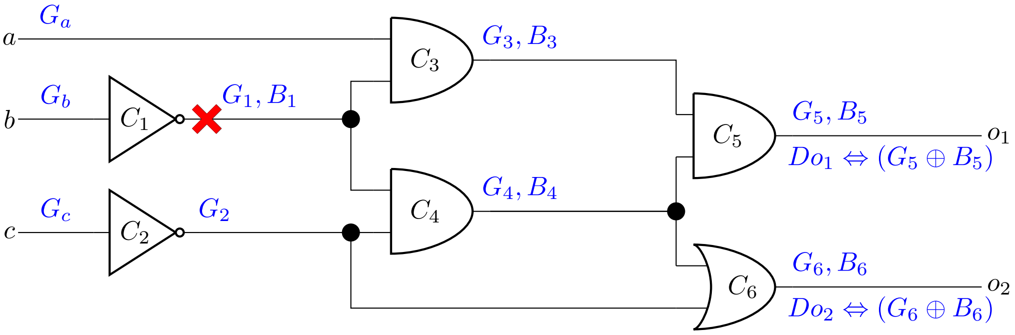


Figure 3.4.: Modeling of the effects of a stuck-at fault in a classic SAT-based ATPG.

The basic description sketched in Figure 3.4 is sufficient for the ATPG algorithm but the solve speed can be improved by adding further information to the formula. All D-chains work by implementing extra reasoning about the difference between the good and bad circuit at every gate in the propagation cone. To this end, additional D variables are added that compute whether there is a difference between the good and bad circuit. They are created similar to the computation of the Do variables at the circuit's outputs.

All of the D-chains that are presented in this thesis are implemented with full support for gates with more than two inputs since these gates allow for a much more efficient encoding of the circuit.

3.2.1. Forward D-Chain

When there is a difference between the good and bad circuit at the output of a gate, this difference will probably propagate to one of the successor nodes of the gate. This assumption is encoded by the forward implication D-chain [39], [54] which is shown in Figure 3.5.

In this circuit, the arrows indicate the implications that are added by the forward D-chain. In detail, the Boolean formula representing the ATPG problem is extended with the sub-formulas shown in Figure 3.6. As previously stated, the forward D-chain is based on the assumption that a difference propagates to a successor gate. This is, however,

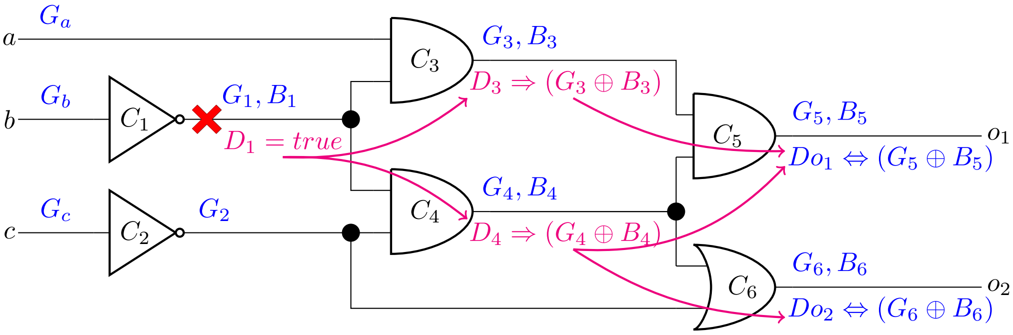

a
b
c
G_a
G_b
G_c
C_1
C_2
G_1, B_1
$D_1 = true$
G_2
C_3
C_4
G_3, B_3
$D_3 \Rightarrow (G_3 \oplus B_3)$
G_4, B_4
$D_4 \Rightarrow (G_4 \oplus B_4)$
C_5
C_6
G_5, B_5
$Do_1 \Leftrightarrow (G_5 \oplus B_5)$
o_1
G_6, B_6
$Do_2 \Leftrightarrow (G_6 \oplus B_6)$
o_2

Figure 3.5.: Implications added by the forward D-chain.

not always the case. Due to re-converging paths in the circuit a difference at the output of a gate might disappear. To allow for this effect, the forward D-chain utilizes a weak difference condition: The D variables for the D-chain are encoded with an implication (see Formulas 3.2 and 3.3) instead of a full equivalence like the Do variables for the outputs. Thus, the solver can always assign the D variables to $false$ in case the difference cannot be propagated.

$$D_1 = true \tag{3.1}$$
$$D_3 \Rightarrow (G_3 \oplus B_3) \tag{3.2}$$
$$D_4 \Rightarrow (G_4 \oplus B_4) \tag{3.3}$$
$$D_1 \Rightarrow (D_3 \vee D_4) \tag{3.4}$$
$$D_3 \Rightarrow Do_1 \tag{3.5}$$
$$D_4 \Rightarrow (Do_1 \vee Do_2) \tag{3.6}$$

Figure 3.6.: List of the sub-formulas that are encoded for the forward D-chain.

At the fault site itself there is always a difference ($D_1 = true$). This further simplifies the formulas – in the presented example, the clause $(D_3 \vee D_4)$ is added directly to the formula because the left side of the implication in Formula 3.4 is definitely $true$.

Cost

For each gate in the propagation cone of the fault, two clauses are required to model the weak difference condition. Furthermore, one additional clause is required to model the forward implication itself. Thus, the total cost per gate is three clauses.

3.2.2. Backward D-Chain

When there is a difference between the good and bad circuit at the output of a gate, there must be a difference on at least one of the inputs of the gate as well. This is the basic idea behind the backward implication D-chain [55] which is shown in Figure 3.7.

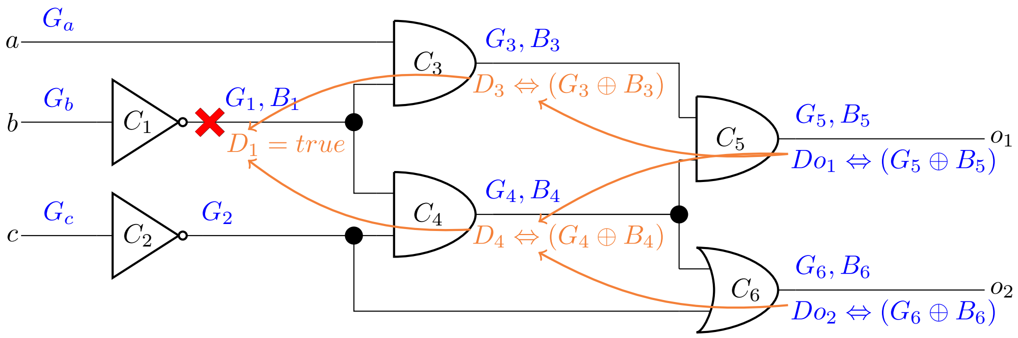

G_a
a
G_b
b
C_1
G_1, B_1
$D_1 = true$
G_c
c
C_2
G_2
C_3
G_3, B_3
$D_3 \Leftrightarrow (G_3 \oplus B_3)$
C_4
G_4, B_4
$D_4 \Leftrightarrow (G_4 \oplus B_4)$
C_5
G_5, B_5
o_1
$Do_1 \Leftrightarrow (G_5 \oplus B_5)$
C_6
G_6, B_6
o_2
$Do_2 \Leftrightarrow (G_6 \oplus B_6)$

Figure 3.7.: Implications added by the backward D-chain.

Again, the arrows indicate the implications that are added by the backward D-chain. In detail, the Boolean formula representing the ATPG problem is extended with the sub-formulas shown in Figure 3.8. Unlike the forward D-chain, the backward D-chain utilizes the normal equivalence for the difference condition (see Equations 3.8 and 3.9). This is possible because a difference between the good and bad circuit does not appear out of nothing. When there is a difference at the output of a gate, there must be a difference on at least one of the gate's inputs.

$$D_1 = true \tag{3.7}$$
$$D_3 \Leftrightarrow (G_3 \oplus B_3) \tag{3.8}$$
$$D_4 \Leftrightarrow (G_4 \oplus B_4) \tag{3.9}$$
$$Do_1 \Rightarrow (D_3 \vee D_4) \tag{3.10}$$
$$Do_2 \Rightarrow D_4 \tag{3.11}$$
$$D_3 \Rightarrow D_1 \tag{3.12}$$
$$D_4 \Rightarrow D_1 \tag{3.13}$$

Figure 3.8.: List of the sub-formulas that are encoded for the backward D-chain.

Cost

For each gate in the propagation cone of the fault, four clauses are required to model the difference condition. Furthermore, one additional clause is required to model the backward implication itself. Thus, the total cost per gate is five clauses.

3.2.3. Combined Backward-Forward D-Chain

The backward and forward D-chains can be combined to give the solver the maximum amount of information about the propagation of the difference. This is achieved by introducing two variables per gate: Df encodes the forward D-chain, whereas Db is responsible for the backward D-chain. In the combined backward-forward D-chain, the weak equivalence condition for the forward D-chain can be encoded more cheaply by simply adding the single clause $Df \Rightarrow Db$ instead of $Df \Rightarrow (G \oplus B)$. This is possible because Db and $(G \oplus B)$ are equivalent. Thus, combining the backward and forward D-chains is slightly cheaper than the sum of the costs for each individual D-chain.

Cost

For each gate in the propagation cone of the fault, four clauses are required to model the difference condition for the backward D-chain. In addition, one clause is needed for the weak difference of the forward D-chain. Furthermore, one additional clause each is required to model the backward and forward implications themselves. Thus, the total cost per gate is seven clauses.

3.3. Good-Diff D-Chain

The backward and forward D-chains each add a new variable for every gate in the propagation cone of the fault which models the difference between the good and bad version of the circuit. Hence, every gate output is represented by three variables: G, B and D. This creates overhead since any two of these three variables can be used to compute the third:

$$D \Leftrightarrow (G \oplus B) \tag{3.14}$$

$$G \Leftrightarrow (B \oplus D) \tag{3.15}$$

$$B \Leftrightarrow (G \oplus D) \tag{3.16}$$

The good-diff D-chain attempts to create a smaller representation of the signal values while still reasoning about the difference like the other D-chains do. To this end, it eliminates the representation of the bad circuit and, instead, stores only the fault-free value, G, of each signal and whether there is a difference to the bad version, D. While this reduces the number of variables per gate output from three to two, it also requires a new gate encoding to accurately compute whether a difference will propagate through the gate. In [56] an indirect two variable circuit encoding similar to the good-diff D-chain was presented. However, while the approach presented in this thesis is applicable to all kinds of cells, the indirect encoding of [56] is limited to basic cells with two inputs.

Figure 3.9 shows the circuit with the variables required for the good-diff D-chain. The arrows indicate the new encoding of the gates.

3.3.1. Gate Encoding

The gate encoding for the good values in the good-diff D-chain is unchanged and is performed with the Tseitin transformation. For the difference literals, the encoding becomes

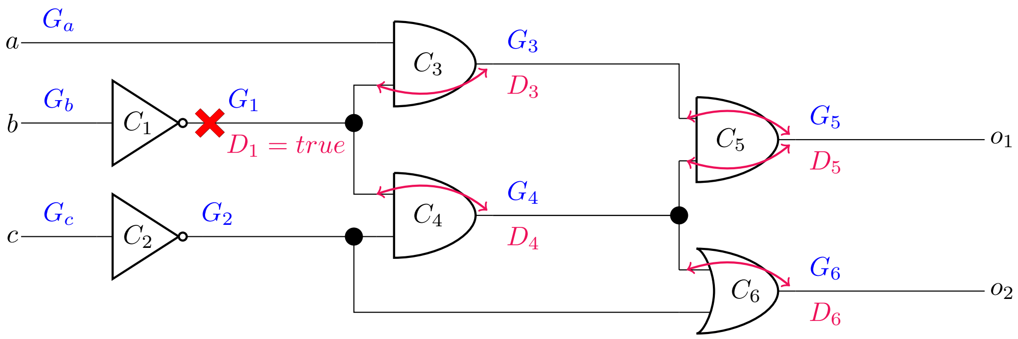


Figure 3.9.: Modeling of the effects of a stuck-at fault with the good-diff D-chain.

more challenging and has to be computed for every gate type. This section gives examples for the encoding of the difference for an AND-gate and for an XOR-gate. The encoding of the remaining standard gates can be derived in a similar manner.

To compute the difference Do at the output of the gate, the good and difference values at all of the gate inputs are required (G_i and D_i, respectively). In the examples, it is assumed that there is a difference variable for every gate input. If the gate is at the border of the fault propagation cone (in the example in Figure 3.9 this is the case for C_3, C_4 and C_6) some inputs cannot have a difference. In this case, the D_i variables corresponding to these inputs are definitely $false$ and the derived clauses can be simplified accordingly.

Difference for an AND-Gate

Table 3.1 shows the value of Do depending on the input values in a two-input AND-gate. The table is generated by first deriving the B values at the inputs, then computing the B value of the output and finally comparing this value to the correct output value.

Table 3.1.: The difference at the output, Do, depending on the input values of a two-input AND-gate.

	No dif.				Dif. at 1				Dif. at 2				Dif. at both			
G_1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
G_2	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
D_1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
D_2	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Do	0	0	0	0	0	0	1	1	0	1	0	1	1	0	0	1

The function table then has to be transformed into a formula in CNF as efficiently as possible. This is performed in two steps: First, each column of the table is transformed into CNF. Then, the columns are optimized through common inference rules for Boolean logic [59]. Figure 3.10 specifies the clauses that are derived directly from the logic table. Figure 3.11 shows the optimized list of clauses.

Clearly, simple logic optimizations greatly reduce the size of the encoding – from 16 clauses containing 80 literals to 9 clauses containing 35 literals. In the formula generated

$$(\neg Do \vee G_1 \vee G_2 \vee D_1 \vee D_2) \tag{3.17}$$
$$(\neg Do \vee \neg G_1 \vee G_2 \vee D_1 \vee D_2) \tag{3.18}$$
$$(\neg Do \vee G_1 \vee \neg G_2 \vee D_1 \vee D_2) \tag{3.19}$$
$$(\neg Do \vee \neg G_1 \vee \neg G_2 \vee D_1 \vee D_2) \tag{3.20}$$
$$(\neg Do \vee G_1 \vee G_2 \vee \neg D_1 \vee D_2) \tag{3.21}$$
$$(\neg Do \vee \neg G_1 \vee G_2 \vee \neg D_1 \vee D_2) \tag{3.22}$$
$$(Do \vee G_1 \vee \neg G_2 \vee \neg D_1 \vee D_2) \tag{3.23}$$
$$(Do \vee \neg G_1 \vee \neg G_2 \vee \neg D_1 \vee D_2) \tag{3.24}$$
$$(\neg Do \vee G_1 \vee G_2 \vee D_1 \vee \neg D_2) \tag{3.25}$$
$$(Do \vee \neg G_1 \vee G_2 \vee D_1 \vee \neg D_2) \tag{3.26}$$
$$(\neg Do \vee G_1 \vee \neg G_2 \vee D_1 \vee \neg D_2) \tag{3.27}$$
$$(Do \vee \neg G_1 \vee \neg G_2 \vee D_1 \vee \neg D_2) \tag{3.28}$$
$$(Do \vee G_1 \vee G_2 \vee \neg D_1 \vee \neg D_2) \tag{3.29}$$
$$(\neg Do \vee \neg G_1 \vee G_2 \vee \neg D_1 \vee \neg D_2) \tag{3.30}$$
$$(\neg Do \vee G_1 \vee \neg G_2 \vee \neg D_1 \vee \neg D_2) \tag{3.31}$$
$$(Do \vee \neg G_1 \vee \neg G_2 \vee \neg D_1 \vee \neg D_2) \tag{3.32}$$

Figure 3.10.: List of the clauses derived by parsing the function table.

$$(\neg Do \vee D_1 \vee D_2) \tag{3.33}$$
$$(\neg Do \vee G_1 \vee D_1) \tag{3.34}$$
$$(\neg Do \vee G_2 \vee D_2) \tag{3.35}$$
$$(\neg Do \vee G_1 \vee \neg G_2 \vee \neg D_2) \tag{3.36}$$
$$(\neg Do \vee \neg G_1 \vee G_2 \vee \neg D_1) \tag{3.37}$$
$$(Do \vee \neg G_1 \vee D_1 \vee \neg D_2) \tag{3.38}$$
$$(Do \vee \neg G_2 \vee \neg D_1 \vee D_2) \tag{3.39}$$
$$(Do \vee \neg G_1 \vee \neg G_2 \vee D_1 \vee D_2) \tag{3.40}$$
$$(Do \vee \neg G_1 \vee G_2 \vee \neg D_1 \vee \neg D_2) \tag{3.41}$$

Figure 3.11.: List of optimized clauses that encode when a difference is propagated to an output of the AND-gate.

by the optimizations, clause 3.33 (which can be re-written as $Do \Rightarrow (D_1 \vee D_2)$) encodes the simple property that a difference at the output requires at least one of the inputs to be different. This corresponds to the idea of the backward D-chain, which can therefore be considered to be a part of the good-diff encoding. However, unlike the clauses encoding the backward D-chain, these clauses cannot be removed from the formula without affecting its correctness.

Difference for an XOR-Gate

The clauses encoding the propagation of the difference to the output of an XOR-gate can be created similar to those for the AND-gate. After optimization, the clauses shown in Figure 3.12 are derived. Here, the difference at the output is completely independent from the good values at the gate inputs. The output differs between the good and the bad circuit if exactly one of the inputs differs.

Cost

Unlike the previously discussed D-chains, the cost of the good-diff D-chain is variable, depending on the gate type and the number of inputs that can have a difference. A two-input AND-gate for example requires 9 clauses with 35 literals if both inputs can show a difference. If only one input can show a difference because the gate is at the border of the

$$(\neg Do \vee D_1 \vee D_2) \tag{3.42}$$
$$(\neg Do \vee \neg D_1 \vee \neg D_2) \tag{3.43}$$
$$(Do \vee D_1 \vee \neg D_2) \tag{3.44}$$
$$(Do \vee \neg D_1 \vee D_2) \tag{3.45}$$

Figure 3.12.: List of optimized clauses that encode when a difference is propagated to an output of the XOR-gate.

fault propagation cone, the costs drop to 3 clauses with 7 literals.

For standard gates with more than two inputs, the cost increases steeply because the encoding becomes much more difficult and there are many more possibilities. An AND-gate with four inputs, for example, requires 36 clauses already, if every input can have a difference. This way, the overall cost in terms of number of clauses is different for every circuit and every fault location.

If the ATPG algorithm would be restricted to two-input gates (by mapping basic cells with more than two inputs to multiple gates) this drawback of the good-diff D-chain in comparison to the other D-chains would be less pronounced. The indirect encoding of [56] is limited to such two-input cells. However, while this encoding requires slightly fewer clauses (27 instead of 36), it adds two more variables and a new logic level for the output of the new intermediate cells. Therefore, this thesis instead presents two novel hybrid D-chains in the next section to tackle the problem of gates where many inputs can have a difference. These hybrid D-chains reduce the number of clauses without adding more variables for every gate. This solution also allows for the efficient integration of different D-chains into the same ATPG since the basic circuit is always efficiently encoded.

3.4. Hybrid D-Chains

The previously introduced good-diff D-chain is very cheap for gates at the border of the fault propagation cone because many clauses do not need to be included. At the same time, for a gate where many inputs can potentially have a difference, the cost is higher than for the traditional encoding – even though fewer variables are required.

The hybrid D-chain concept attempts to reduce the overall cost by combining the best aspects of the conventional and the good-diff encoding. This is achieved by selectively re-introducing the bad value for some signals in the otherwise good-diff encoded circuit. These bad values are then used to encode some of the gates in the conventional manner.

Figure 3.13 shows an example for a hybrid D-chain. Here, gate C_5 is modeled conventionally. Hence, the bad values at its inputs, B_3 and B_4, have to be provided. Furthermore, the D_5 value at the output has to be computed again from the G_5 and B_5 value. In this small example, the cost for the hybrid D-chain is actually slightly higher (12 clauses) than the good-diff encoding alone (9 clauses). However, in real circuits the resulting formula might be smaller if the right gates are encoded conventionally. A backward D-chain is added to all conventionally modeled gates to allow the solver to keep reasoning about the propagation of the difference in all gates.

The selection process for conventionally modeled gates is performed by a heuristic. Two different heuristics are implemented for the evaluation.

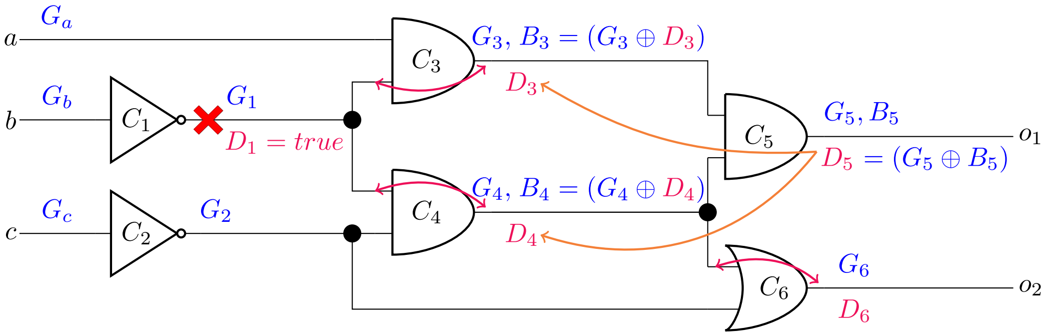


Figure 3.13.: Modeling of the effects of a stuck-at fault with a hybrid D-chain.

3.4.1. Static Node Selection Heuristic

The static node selection heuristic models all gates where more than one input can have a difference in the conventional manner. Thus, the good-diff encoding is used for the gates at the border of the fault propagation cone but not for the remaining gates in the cone. Figure 3.14 indicates the chosen modeling depending on the position of a gate in the cone. For outputs which are not covered by the good-diff encoding, the difference D is computed with an XOR-gate again.

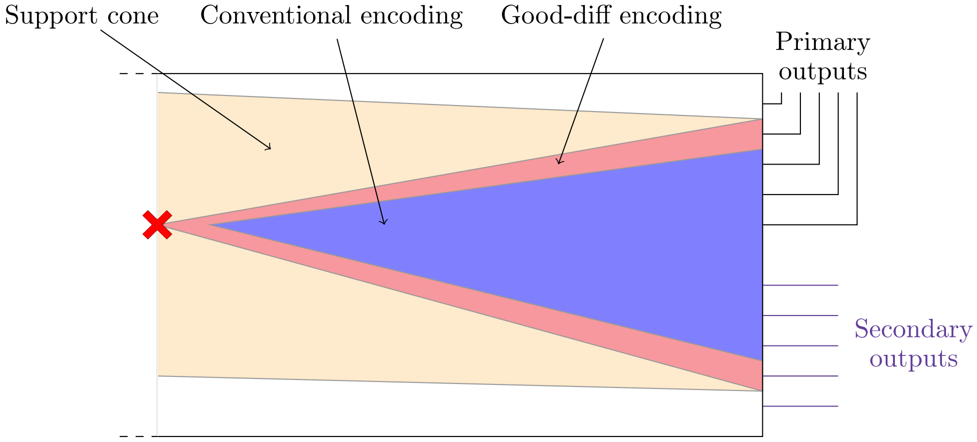


Figure 3.14.: The areas of the fault propagation cone that are modeled conventionally and with the good-diff encoding for the static node selection heuristic.

3.4.2. Dynamic Node Selection Heuristic

The dynamic node selection heuristic uses a three step approach to identify nodes that are to be modeled conventionally. The general idea is to create the B values only in cases where many D variables of inputs would have to be used.

In the first step, the number of possible differences at the inputs of each gate is counted. This gives the score of the gate. Next, the combined successor score (*css*) is computed for each gate – this is the sum of the scores of the successor gates the current gate's output is connected to. If a gate has a *css* of two or higher, it is marked. A marked gate has to provide the B value at the output. This can be achieved in two ways: Firstly, the B value can be re-created from the G and D variables or, alternatively, the gate can be modeled in the conventional manner.

In the final step, the actual decision with regard to the modeling is made. A node is modeled conventionally if it has at least two inputs that can show a difference and if for either all or all but one of these inputs a B value is available. When a gate is modeled conventionally, it also provides a B value at its output. Thus, it might occur that the conventional modeling threshold for one of its successor gates is exceeded. Therefore, the last step is repeated until a fix point has been reached, but at most three times. All nodes that are not modeled conventionally are encoded with the good-diff D-chain.

Figure 3.15 shows the effect of the dynamic heuristic. The gates are annotated with their score and *css*. The gates C_3 and C_4 each have a *css* of at least two and are, therefore, providing a B value at their output. Gate C_5 is modeled conventionally because it has two inputs that can show a difference and two B values available. On the other hand, gate C_6 has only one input that can show a difference and is, thus, modeled with the good-diff encoding (which is much cheaper if only a single input can show a difference).

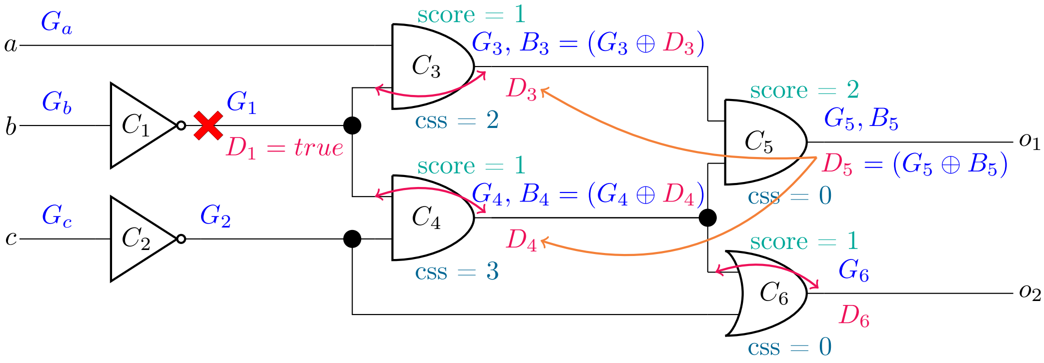


Figure 3.15.: Node modeling by the dynamic node selection heuristic.

3.5. Evaluation

This chapter discussed six different D-chains, three of which are based on a novel concept of modeling the good and bad circuit. In addition, there is of course always the option of not adding a D-chain at all. In this section, a thorough evaluation of the different D-chains is performed across a wide selection of circuits.

For the first set of experiments the incremental solving mode of the ATPG is disabled to give a clear picture of the gains of the D-chains.

Normally, ATPG algorithms utilize fault simulation to quickly determine if a test pattern detects more than just the one fault that it was originally created for. If a fault is detected by any test pattern, it is removed ("dropped") from the list containing all possible faults in the circuit (the fault list) and not considered further.

Since the goal of this chapter lies with the development and analysis of D-chains for the ATPG algorithm, a fault simulator is not used for most of the experiments. This allows for the exact evaluation of the effect of the different D-chains across all possible stuck-at faults without any faults being dropped by chance. Furthermore, since each D-chain will most likely produce different test-patterns for the same fault, different faults would be dropped through fault simulation. Thus, after the first fault each ATPG instance would work on a different fault set and the results would contain a large amount of random variation. To allow for a scientifically accurate comparison of the different D-chains, fault simulation is generally not used during the evaluation.

The evaluation is structured as follows: In Section 3.5.1 the general ATPG performance without D-chains is shown. Next, Section 3.5.2 evaluates the gains of the different D-chains for all of the considered circuits and gives an in-depth analysis of some of the performance parameters. In the subsequent Section 3.5.3 the incremental solving mode of the ATPG is enabled and the resulting change in solve speed is analyzed.

To show the influence of the D-chains in a more practical and realistic environment the experiments are repeated with a fault simulator developed by Pascal Raiola with results shown in Section 3.5.4. These results are only meant to validate that D-chains are useful in SAT-based ATPG and not as an exact comparison.

In SAT-based ATPG the total runtime is mainly influenced by two factors [47]: The time that is required to generate a formula and the time that is required to solve that formula – the solve time. D-chains are a technique to improve the solve time by guiding the SAT solver. Therefore, this value is the focus of the following evaluations.

All of the tables referenced in the evaluation are printed at the end of the chapter.

For all experiments the previously described SAT-based stuck-at ATPG is used. As SAT solver, `antom` [13], [14] with a timeout of 10 s per fault is utilized.

3.5.1. Without D-Chain

Table 3.2 shows an overview of the number of stuck-at faults, solve time, overall memory consumption and achieved fault coverage (FC) as well as the number of timeouts for each of the circuits when no D-chain is used. Since all of the circuits are assumed to be full-scan, the ATPG can achieve a very high fault coverage and a pattern can be computed for almost every fault within 10 seconds. It should be noted that the ATPG produces a new pattern for every single fault – there is no fault simulation at all. Thus, while the overall solve time is close to 1 hour for some of the largest circuits, the average solve time per fault is still well below 100 ms for every circuit. Figure 3.16 shows the average solve time and formula generation time per fault.

The solve time strongly depends on the difficulty of testing the faults, whereas the generation time is mostly influenced by the circuit's structure and size. On some circuits, the solve time greatly outweighs the generation time (e.g., AES 10-2-4-4_d). On others, the situation is reversed. It should also be noted that the solve time can vary widely between different faults for the same circuit. Consider the circuit *vga_lcd* which has an average solve time of below 10 ms. Nonetheless, 57 timeouts (after 10 s) occurred. These hard-to-detect faults are of special interest when optimizing an ATPG algorithm and will be analyzed further in Section 3.5.2.

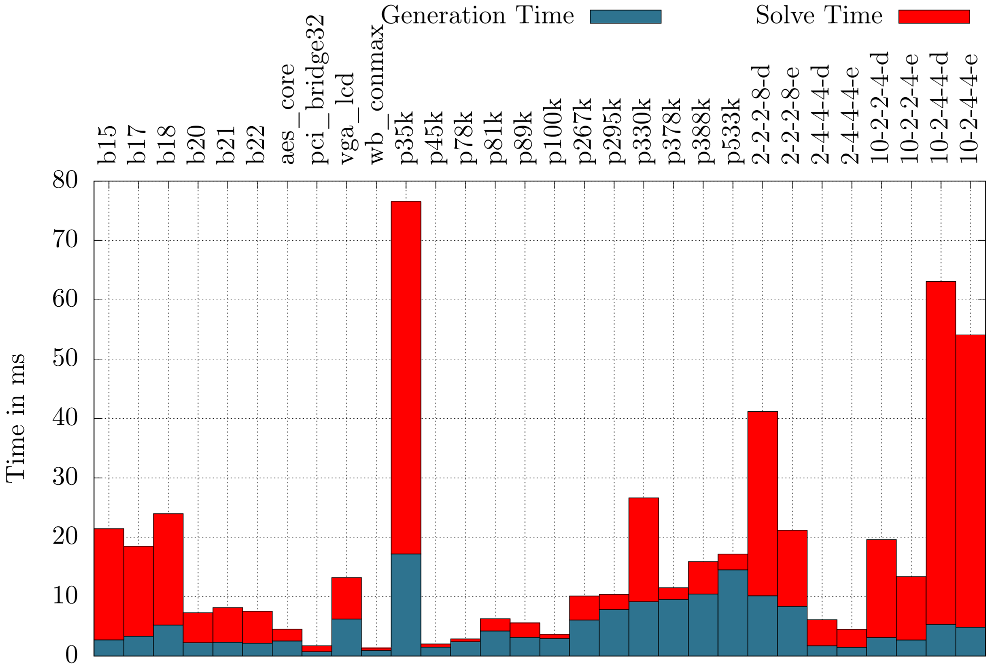


Figure 3.16.: Average **formula generation** and **solve time** per fault without D-chains or incremental solving.

3.5.2. With D-Chains

To evaluate the different D-chains, the test pattern generation is repeated once for every D-chain and circuit. Table 3.3 shows the change in solve time compared to the ATPG without any D-chains. The results are visualized in Figure 3.17 and summarized per circuit group in Figure 3.18.

The first major observation is that usually the solve speed is dramatically increased when a D-chain is added. This holds true for the vast majority of D-chain circuit combinations. Interestingly, the earliest suggested D-chain, the forward D-chain introduced in [39], [54], is the only D-chain that is generally not beneficial to the solve speed. This might be because modern SAT solvers can deduce the knowledge provided by the forward D-chain more cleverly on their own.

The second observation is that for most circuit groups the gains by the remaining D-chains are very similar. The average decrease in solve time is about 70-90 %. This general observation is, however, not accurate for the cryptographic AES circuit benchmarks. Here, the hybrid D-chain with the dynamic heuristic and, to an extend, the normal good-diff D-chain vastly outperform the remaining D-chains. They reduce the solve time by about 55 % on average, compared to only 17 % for the backward D-chain as the best classic D-chain variant.

Overall, across all different circuits the hybrid D-chain with the dynamic heuristic provides the fastest solve times and consistent speedups for every circuit. Nonetheless, depending on the circuit other D-chains might be even better suited. For an optimal solve speed in a specific application or for a specific class of circuits, a comparison of the different D-chain techniques should be performed.

Timeouts

The average solve time for the considered faults is well below 100 ms as Figure 3.16 clearly showed. Nonetheless, even with a timeout of 10 s not every fault can be successfully characterized. This means that there are a couple of faults for which it is more than 100 times harder to find a test pattern than it is for the average fault. Finding a test pattern for these faults is significant because their difficulty means that they are unlikely to be found by a simple random pattern. Alternatively, the fault might be untestable and should be ignored for any further fault coverage optimization.

D-chains not only increase the average solve speed as was highlighted in the previous section, but they also improve the solve speed for these hard-to-detect faults. This is evident from Figure 3.19 which shows the total number of timeouts for the different D-chains.

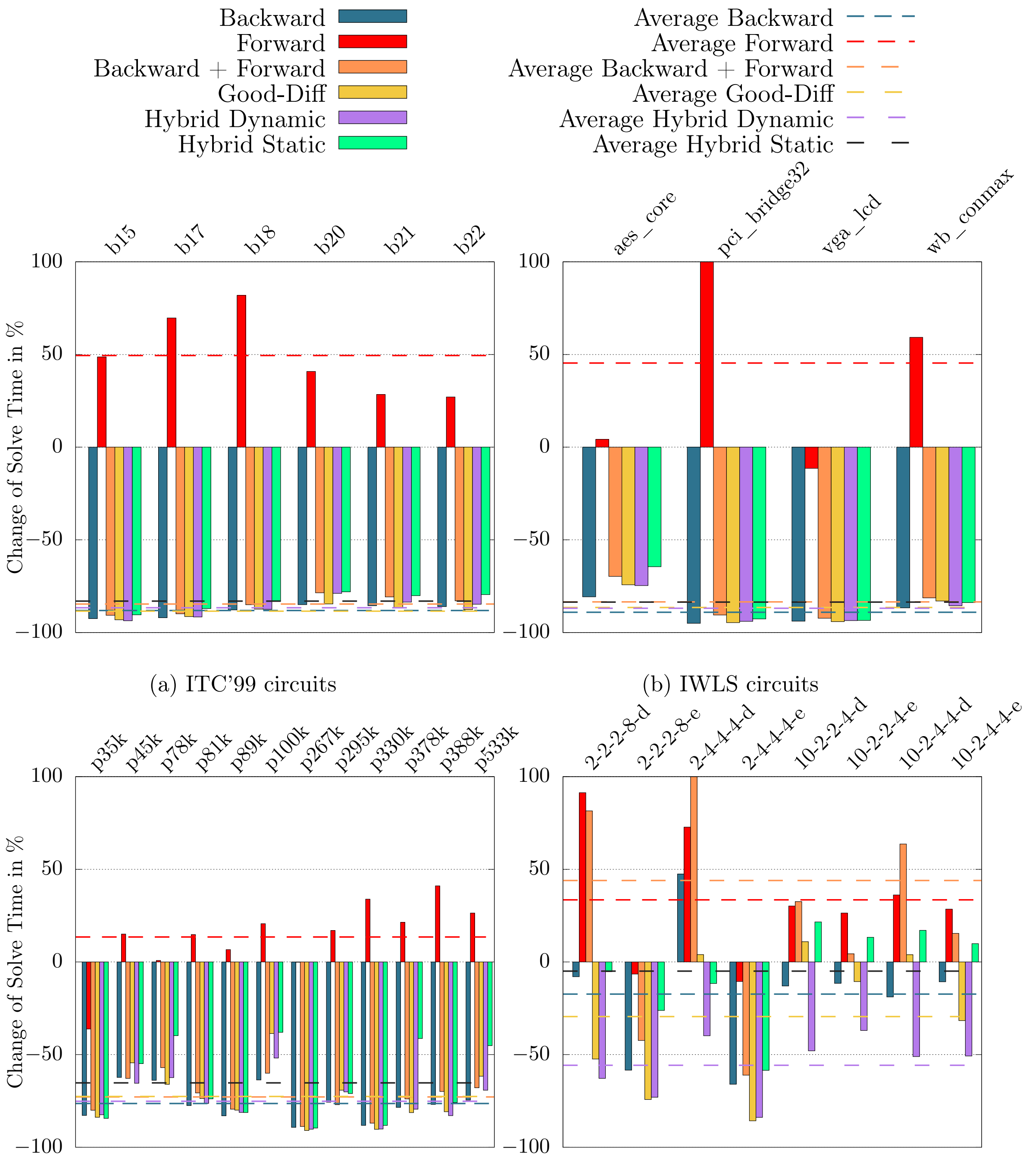
The total number of timeouts is drastically reduced for all different circuit groups when adding a D-chain. Only for the complex industrial benchmark circuits from NXP some timeouts remain. The newly developed good-diff D-chain and especially the hybrid D-chain with the dynamic heuristic prove to be well suited for the characterization of hard-to-detect faults. While the backward D-chain provides slightly lower average solve times than any of the good-diff D-chains on the NXP circuits, all good-diff variants clearly outperform it on the hard-to-detect faults, reducing the number of timeouts by almost another 50 % in direct comparison.

Formula Size

Adding supplemental information to a formula increases its size. Figure 3.20 shows the average increase in formula size measured in the number of clauses.

Curiously, the good-diff encoding and its variants – which were developed to reduce the size of the D-chains by removing redundancies – are actually comparable to the backward D-chain and larger than the forward D-chain. Furthermore, the hybrid versions are sometimes larger than the original good-diff D-chain. Nonetheless, the experimental results show that the good-diff encoding provides substantial benefits and is often faster than the backward D-chain.

Thus, there appears to be no relation between the achieved solve speed and the formula size. Overall, the results show that the inclusion of additional helpful (but potentially redundant) information clearly helps the SAT solver. On the other hand, not all information is actually helpful. The forward D-chain shows the smallest increase in formula size but actually slows down the solver.



(a) ITC'99 circuits

(b) IWLS circuits

(c) NXP circuits

(d) AES circuits

Figure 3.17.: Change in **solve time** for each circuit with the different D-chain variants compared to utilizing no D-chain at all.

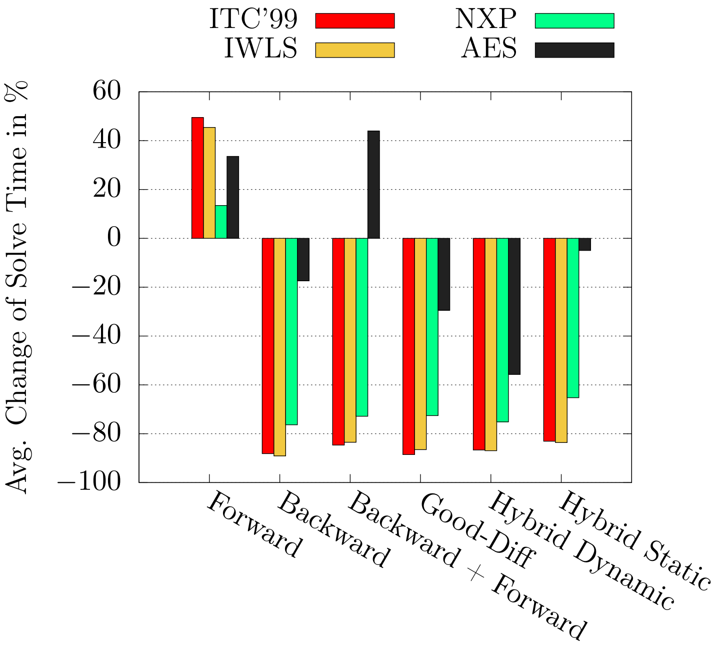


Figure 3.18.: Average change in **solve time** for the different D-chain variants per circuit class.

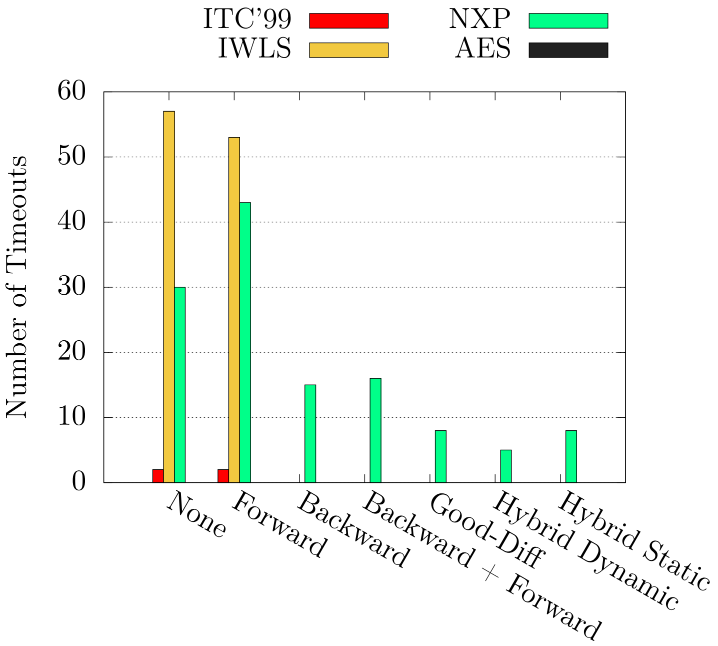


Figure 3.19.: Total number of **timeouts** per circuit class for each D-chain.

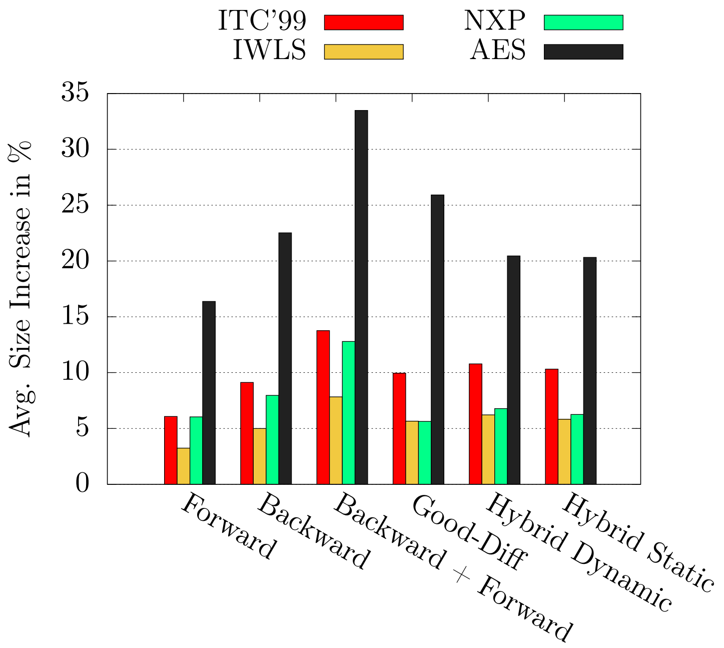


Figure 3.20.: Average increase in **formula size** (measured in the number of clauses) depending on the circuit class and D-chain.

3.5.3. Incremental Solving

Through incremental solving the formula is created in a way such that the fault has to be visible at exactly one of the outputs in the fault propagation cone. Thus, the number of possibilities for the fault propagation are greatly reduced. Nonetheless, there might still be a large number of different paths through the propagation cone, each of which might carry the fault effect to the output.

To evaluate the effectiveness of D-chains in combination with incremental solving, the exact same circuits and stuck-at faults as before are analyzed again. The results are summarized in Figure 3.21 and are shown in detail in Table 3.4, which lists the initial solve times without a D-chain and the change in solve time for the different D-chain variants.

Generally, solving the formula incrementally greatly decreases the solve time until a test pattern for a fault is found. For easy-to-test faults where the fault effect can be propagated to many outputs, the incremental approach quickly guides the solver towards a solution. Conversely, for hard-to-detect faults an overhead is incurred because the formula is built up incrementally in many smaller steps. This is especially true for untestable faults – here the solver has to traverse every single iteration before the fault is proven to be untestable. In the considered examples with a very high testability, the latter point is negligible. Nonetheless, for some circuits the total solve time with incremental solving is worse than it was with the classic approach.

Overall, the effect of D-chains becomes slightly less pronounced because of the often very simple incremental formulas which only contain the propagation to a single output. However, for most considered circuits adding a D-chain is still beneficial for the total solve time. Across all circuits, the hybrid D-chain with the dynamic node selection heuristic provides the greatest benefit with an average 22.9 % decrease in solve time.

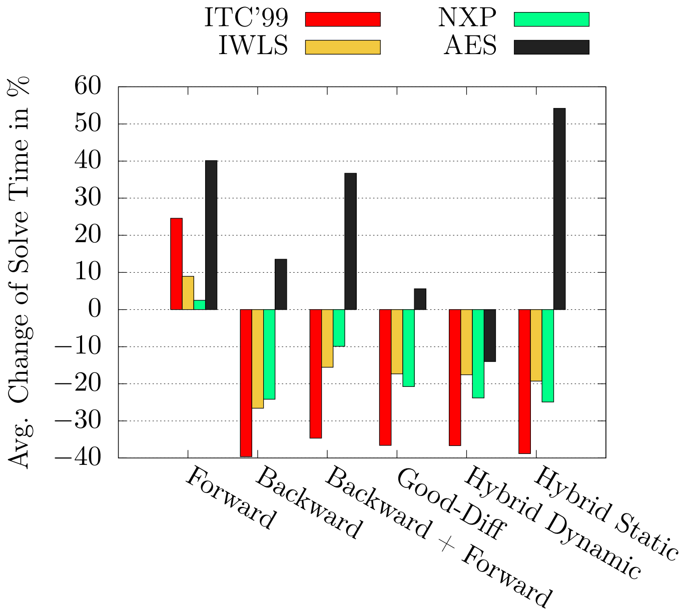


Figure 3.21.: Average change in **solve time** for the different D-chain variants grouped by the different circuit classes with the incremental solving mode.

3.5.4. Fault Simulation

For the final experiments a fault simulator is added to the ATPG. Once a test pattern for a fault is found by the ATPG, the simulator checks if this pattern detects any of the – as of yet – undetected faults. If so, this fault is marked as detected and no additional test pattern to detect the fault is created. Simulation can reduce both the ATPG runtime as well as the number of required test patterns by orders of magnitude and is, therefore, commonly used by ATPG tools.

However, for the analysis of D-chains, simulation might influence the results in a way that does not allow for any valid conclusions. For an example consider a stuck-at fault f in a circuit. Assume that the test pattern "0011" is found by the ATPG without D-chains whereas the ATPG with the backward D-chain finds the test pattern "0110". Two different test patterns for the same fault are not unusual and since a SAT solver makes decisions based on a heuristic, the differences between the two formulas can easily yield this outcome. Assume further that "0011" does not detect any other fault while "0110" detects 10 so far undetected faults. As a result, depending on the total number of faults, the overall solve time of the two modes might differ significantly. Clearly, such random influences are highly problematic when analyzing and comparing different approaches. Therefore, fault simulation was not considered for any of the previous experiments.

The experimental results in this section are meant to show that D-chains are actually applicable to real world ATPG scenarios where fault simulation is used.

Table 3.5 shows the solve time as well as the number of SAT solver calls for the different D-chain variants when fault simulation is used. Note that the SAT solver is used incrementally and might be called many times for a single fault. Clearly, the number of solver calls differs between the different D-chain modes. Consider, for example, the ATPG for the circuit $p35k$. Without any D-chains the solver is called only 5461 times, whereas some D-chain variants result in up to 1000 additional solver calls. While, in this instance, not utilizing any D-chain requires the least amount of SAT solver calls, on other circuits (e.g., $b15$) the situation is reversed.

Thus, the previously predicted random influence – due to different test patterns being found for the same fault – is observed in practice as well. Hence, the average change in solve time for the different D-chains that is summarized in Figure 3.22 should only be seen as an indication and most likely contains some noise.

Nonetheless, the results do show that D-chains provide a great improvement to the solve speed, even when both fault simulation and incremental solving are used.

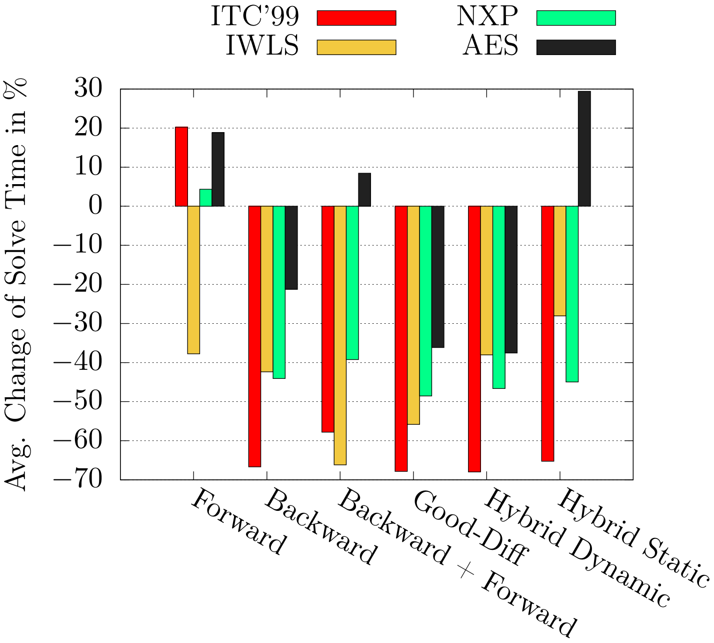


Figure 3.22.: Average change in **solve time** for the different D-chain variants grouped by the different circuit classes when utilizing fault simulation.

3.6. Summary

This chapter introduced, discussed and evaluated different D-chain variants as an improvement to SAT-based ATPG. These include the classic forward and backward D-chain as well as the good-diff encoding and novel hybrid D-chains which reduces the amount of redundancies. The key points with regard to improving SAT-based techniques in circuit test, demonstrated by the thorough evaluation, are the following:

- In general, D-chains greatly improve the performance of the SAT solver especially for hard-to-detect faults.
- The performance of a particular D-chain depends on the circuit (class). Nonetheless, the hybrid D-chain with a dynamic node selection heuristic works very well as a general-purpose D-chain across all of the analyzed circuits.
- The final formula size is not related to the solve time. Creating a more detailed representation of the circuit and of the propagation of the difference between the faulty and fault-free circuit can provide a much larger benefit than the additional clauses are a burden.
- D-chains remain important even when utilizing the SAT-based ATPG algorithm in a highly optimized manner with incremental solving and fault simulation.

Overall, this chapter showed that even relatively minor additions to the formula can be very helpful in guiding the SAT solver. The knowledge and experience obtained in this chapter will be applied to the ATPG algorithms presented in the remainder of this work and should be considered when designing any SAT-based ATPG algorithm.

Table 3.2.: Results of the test pattern generation without a D-chain.

	Circuit	# Faults	Solve Time (s)	Memory (MB)	FC (%)	# Timeouts
ITC'99	b15	14 312	267.72	56.1	99.33	0 (0.00 %)
	b17	43 931	665.41	118.8	98.83	0 (0.00 %)
	b18	119 701	2 244.66	307.7	99.95	2 (0.00 %)
	b20	18 560	93.47	69.1	99.83	0 (0.00 %)
	b21	19 545	114.44	69.8	99.94	0 (0.00 %)
	b22	26 421	142.15	85.9	99.89	0 (0.00 %)
IWLS	aes_core	38 511	75.73	97.3	100.00	0 (0.00 %)
	pci_bridge32	33 191	33.00	107.3	100.00	0 (0.00 %)
	vga_lcd	189 980	1 327.90	472.2	99.97	57 (0.03 %)
	wb_conmax	73 922	29.82	154.2	99.99	0 (0.00 %)
NXP	p35k	34 265	2 032.93	114.0	99.99	0 (0.00 %)
	p45k	42 316	23.46	122.7	99.99	0 (0.00 %)
	p78k	81 367	38.99	226.3	100.00	0 (0.00 %)
	p81k	155 394	320.53	321.7	100.00	1 (0.00 %)
	p89k	96 414	234.37	233.8	99.94	0 (0.00 %)
	p100k	92 176	63.83	246.7	99.95	0 (0.00 %)
	p267k	203 137	823.04	512.7	99.99	0 (0.00 %)
	p295k	240 457	610.60	618.2	99.52	7 (0.00 %)
	p330k	184 867	3 227.25	497.9	99.83	14 (0.01 %)
	p378k	422 389	834.49	1 037.9	100.00	0 (0.00 %)
	p388k	443 483	2 429.06	1 026.2	99.98	3 (0.00 %)
	p533k	692 727	1 840.57	1 569.0	99.87	5 (0.00 %)
AES	2-2-2-8_d	18 795	583.24	70.8	99.99	0 (0.00 %)
	2-2-2-8_e	15 200	194.58	63.2	100.00	0 (0.00 %)
	2-4-4-4_d	6 256	27.46	37.7	100.00	0 (0.00 %)
	2-4-4-4_e	4 888	14.81	35.8	100.00	0 (0.00 %)
	10-2-2-4_d	5 656	93.30	37.2	100.00	0 (0.00 %)
	10-2-2-4_e	5 622	60.13	37.1	100.00	0 (0.00 %)
	10-2-4-4_d	10 586	611.18	46.8	100.00	0 (0.00 %)
	10-2-4-4_e	10 527	518.18	48.7	100.00	0 (0.00 %)

Table 3.3.: Change in solve time (in %) when utilizing the different D-chains compared to the ATPG without any D-chains.

	Circuit	Forward	Backward	Backward	Good-Diff	Hybrid	Hybrid
				+ Forward		Dynamic	Static
ITC'99	b15	48.68	-92.52	-90.70	-93.17	**-93.65**	-90.46
	b17	69.73	**-92.02**	-89.92	-91.39	-91.59	-87.09
	b18	81.90	**-87.79**	-85.03	-87.41	-87.50	-83.18
	b20	40.85	**-84.96**	-78.56	-84.61	-78.92	-78.04
	b21	28.43	-85.52	-80.81	**-86.87**	-83.59	-80.12
	b22	27.11	-86.01	-82.91	**-87.67**	-84.75	-79.51
	Average:	49.45	-88.14	-84.66	**-88.52**	-86.67	-83.06
IWLS	aes_core	4.20	**-80.76**	-69.72	-74.24	-74.70	-64.47
	pci_bridge32	129.60	**-95.02**	-90.56	-94.69	-93.96	-92.67
	vga_lcd	-11.44	-93.88	-92.31	**-94.13**	-93.56	-93.46
	wb_conmax	59.25	**-86.71**	-81.34	-82.99	-85.53	-83.85
	Average:	45.40	**-89.09**	-83.48	-86.51	-86.94	-83.61
NXP	p35k	-36.15	-82.79	-80.01	-83.79	-82.46	**-84.41**
	p45k	15.09	-62.27	-62.82	-54.40	**-65.43**	-54.86
	p78k	0.81	-63.88	-56.99	**-66.02**	-62.37	-39.76
	p81k	14.78	**-77.46**	-70.66	-73.67	-76.31	-73.84
	p89k	6.67	**-83.04**	-79.45	-79.98	-81.15	-81.21
	p100k	20.67	**-63.67**	-60.00	-38.61	-51.86	-37.93
	p267k	-0.05	-89.24	-88.75	**-90.94**	-90.25	-89.59
	p295k	16.94	-75.67	**-77.00**	-69.20	-70.08	-71.00
	p330k	33.88	-88.16	-86.95	**-90.31**	-90.11	-88.18
	p378k	21.41	-78.49	-73.89	**-81.26**	-79.44	-41.26
	p388k	41.06	-76.89	-69.87	-80.80	**-82.92**	-75.84
	p533k	26.42	**-74.54**	-67.86	-61.62	-69.17	-45.12
	Average:	13.46	**-76.34**	-72.85	-72.55	-75.13	-65.25
AES	2-2-2-8_d	91.35	-7.93	81.51	-52.36	**-62.79**	-5.26
	2-2-2-8_e	-6.52	-58.35	-42.35	**-74.21**	-72.99	-26.16
	2-4-4-4_d	72.80	47.47	257.47	3.98	**-39.79**	-11.58
	2-4-4-4_e	-10.53	-65.99	-61.05	**-85.68**	-83.87	-58.48
	10-2-2-4_d	30.18	-12.95	32.56	10.95	**-47.92**	21.60
	10-2-2-4_e	26.40	-11.52	4.41	-10.60	**-36.93**	13.30
	10-2-4-4_d	36.15	-18.87	63.64	3.93	**-51.04**	17.08
	10-2-4-4_e	28.51	-10.69	15.41	-31.63	**-50.72**	9.91
	Average:	33.54	-17.35	43.95	-29.45	**-55.76**	-4.95

Table 3.4.: Initial solve time without any D-chains and change in solve time (in %) when utilizing the different D-chains in the incremental solving mode.

	Circuit	Without	Forward	Backward	Backward	Good-Diff	Hybrid	Hybrid
		(in s)			+ Forward		Dynamic	Static
ITC'99	b15	10.54	15.68	-45.79	-37.40	-49.77	**-53.64**	-50.08
	b17	51.20	68.08	-71.66	-68.29	**-76.44**	-74.56	-72.12
	b18	77.55	32.73	-39.25	-35.66	-45.75	-44.90	**-50.76**
	b20	4.24	9.92	-21.81	**-23.80**	-19.92	-19.64	-21.06
	b21	4.82	12.02	**-27.45**	-21.31	-17.74	-14.01	-22.55
	b22	6.08	9.01	**-31.69**	-21.37	-9.86	-13.08	-16.31
	Average:		24.57	**-39.61**	-34.64	-36.58	-36.64	-38.81
IWLS	aes_core	2.26	7.08	-38.23	**-39.29**	-36.11	-38.58	-34.87
	pci_bridge32	0.54	9.56	**-10.29**	15.44	10.29	13.24	8.82
	vga_lcd	31.99	-3.33	-30.36	-26.41	-24.36	-28.77	**-35.95**
	wb_conmax	2.26	22.52	**-27.48**	-11.88	-19.15	-16.13	-15.07
	Average:		8.96	**-26.59**	-15.53	-17.33	-17.56	-19.27
NXP	p35k	82.17	**-33.81**	-4.92	3.45	8.27	7.00	-4.27
	p45k	5.72	-5.25	**-28.97**	-15.89	-23.02	-14.77	-11.76
	p78k	5.50	6.98	-10.39	-12.57	-18.24	-13.95	**-23.18**
	p81k	**13.70**	40.47	21.05	20.32	41.66	12.15	5.78
	p89k	10.95	32.87	-10.70	-1.61	-5.66	**-16.18**	-8.73
	p100k	13.77	26.17	-19.81	-19.81	-18.50	**-25.53**	-24.34
	p267k	18.01	21.34	0.29	14.19	**-2.73**	1.22	1.13
	p295k	119.52	2.31	-45.51	62.43	-54.75	-56.97	**-57.80**
	p330k	156.73	-42.43	**-43.22**	-32.54	-28.99	-32.79	-40.53
	p378k	32.24	-4.19	-25.15	-21.59	-28.60	-28.76	**-30.77**
	p388k	100.56	5.38	**-28.52**	-22.14	-24.12	-23.03	-23.00
	p533k	7 303.66	-20.19	**-94.05**	-93.18	-94.03	-93.92	-81.38
	Average:		2.47	-24.16	-9.91	-20.73	-23.79	**-24.90**
AES	2-2-2-8_d	308.50	75.31	21.54	44.20	**-80.64**	-79.06	15.06
	2-2-2-8_e	28.13	13.95	-26.86	**-37.78**	-15.24	-23.42	-16.69
	2-4-4-4_d	9.48	26.91	49.64	106.41	-78.41	**-79.97**	-18.47
	2-4-4-4_e	1.18	-3.04	-18.58	-11.15	**-29.39**	-18.58	-19.59
	10-2-2-4_d	**100.74**	55.17	27.77	46.74	51.74	45.56	201.96
	10-2-2-4_e	59.84	42.50	6.83	11.80	37.48	**-20.19**	34.56
	10-2-4-4_d	**862.52**	53.61	37.74	90.27	71.27	52.73	181.63
	10-2-4-4_e	**560.18**	56.62	10.46	43.16	87.97	10.88	55.04
	Average:		40.13	13.57	36.71	5.60	**-14.01**	54.19

Table 3.5.: Solve time (top row, in s) and number of solver calls (bottom row) for the different D-chains when fault simulation is used.

	Circuit	Without	Forward	Backward	Backward	Good-Diff	Hybrid	Hybrid
					+ Forward		Dynamic	Static
ITC'99	b15	2.49	3.07	0.73	0.80	**0.48**	0.53	0.57
		9 653	9 538	9 440	9 319	9 554	9 561	9 528
	b17	11.70	19.43	2.44	2.94	**1.87**	2.12	2.23
		26 606	26 430	27 019	27 191	27 759	27 951	27 670
	b18	9.90	10.54	1.86	2.12	**1.64**	1.67	1.96
		8 908	8 914	9 842	9 982	9 602	9 850	9 709
	b20	0.47	0.48	0.23	0.32	0.22	**0.21**	0.24
		1 739	1 775	1 737	1 759	1 759	1 759	1 759
	b21	0.61	0.62	0.28	0.32	**0.27**	**0.27**	0.28
		1 634	1 663	1 695	1 688	1 728	1 678	1 686
	b22	0.75	0.91	**0.27**	0.40	0.38	0.35	0.37
		1 848	1 882	1 947	1 919	1 865	1 878	1 893
IWLS	aes_core	0.03	0.02	**0.01**	0.02	0.02	**0.01**	0.02
		637	636	645	642	647	647	647
	pci_bridge32	0.03	0.01	0.03	**0.00**	0.01	**0.00**	0.03
		873	860	860	847	867	867	860
	vga_lcd	0.33	0.27	0.26	**0.19**	0.24	**0.19**	0.20
		4 968	5 030	5 027	5 038	5 065	5 051	5 064
	wb_conmax	0.03	0.02	**0.00**	**0.00**	0.01	0.04	0.02
		659	673	638	629	671	649	644
NXP	p35k	12.48	13.20	**11.32**	13.39	12.64	11.56	11.42
		5 461	5 554	6 404	6 365	6 397	6 424	6 394
	p45k	0.95	0.76	**0.44**	0.52	0.54	0.52	0.50
		2 127	2 088	2 191	2 228	2 149	2 225	2 145
	p78k	0.00	0.00	0.00	0.00	0.00	0.01	0.00
		96	94	98	94	99	94	100
	p81k	0.98	0.92	**0.77**	0.94	0.82	0.82	0.80
		10 928	10 862	11 156	11 151	11 083	11 096	11 068
	p89k	0.84	0.90	0.62	0.68	**0.58**	0.60	0.62
		5 793	5 870	6 153	6 018	5 995	6 149	6 141
	p100k	1.00	1.22	0.47	0.48	0.49	**0.44**	0.53
		2 688	2 707	2 697	2 724	2 682	2 717	2 697
	p267k	1.07	1.33	0.60	0.69	0.56	0.60	**0.47**
		8 016	7 926	8 083	8 068	8 044	8 177	7 981
	p295k	17.95	14.26	5.20	5.17	4.52	4.75	**4.48**
		38 548	38 472	38 462	38 430	38 463	38 558	38 491
	p330k	19.39	24.63	12.61	12.71	**11.14**	11.38	12.41
		14 898	14 598	14 764	14 934	14 953	15 163	15 048
	p378k	0.02	0.02	0.01	0.01	**0.00**	0.01	0.01
		189	187	188	184	185	181	183
	p388k	1.99	2.59	1.32	1.62	1.01	1.14	**0.99**
		9 414	9 314	9 529	9 288	9 479	9 259	9 361
	p533k	144.59	112.92	**2.26**	2.40	**2.26**	2.67	15.89
		15 105	15 192	15 343	15 226	15 316	15 178	15 373
AES	2-2-2-8_d	18.67	23.93	18.61	25.58	5.27	**4.57**	25.19
		950	984	1 063	1 071	1 266	1 334	1 074
	2-2-2-8_e	3.17	3.51	**1.42**	1.70	2.00	1.51	1.72
		1 022	1 048	1 244	1 244	1 268	1 270	1 249
	2-4-4-4_d	0.36	0.53	0.55	0.92	0.08	**0.05**	0.37
		97	92	112	98	176	163	111
	2-4-4-4_e	0.12	0.09	0.04	0.08	0.05	0.08	**0.04**
		226	210	249	250	254	263	258
	10-2-2-4_d	1.57	1.95	**1.29**	1.30	1.40	1.35	4.78
		58	59	59	57	55	55	58
	10-2-2-4_e	0.91	0.89	0.48	0.90	0.64	**0.37**	0.93
		57	53	64	61	59	56	60
	10-2-4-4_d	5.87	9.84	**4.63**	6.11	6.54	8.50	11.52
		65	66	63	61	64	67	68
	10-2-4-4_e	3.56	3.53	2.85	**2.21**	3.10	2.58	3.90
		60	65	63	63	60	63	65

4. Testing Transistor Stuck-Open Faults

A SAT-based ATPG approach allows for a large versatility regarding the fault model and extra conditions that might restrict a test pattern. Furthermore, many general improvements – like the advanced D-chains presented in the previous chapter – can be applied to almost any SAT-based ATPG algorithm.

The versatility and extendibility of SAT-based test pattern generation approaches is used in this chapter to develop a transistor stuck-open fault (TSOF) ATPG algorithm. Unlike the simple stuck-at fault model, the TSOF model is more closely related to real physical defects. Although classically assumed to be tested by transition-delay fault tests, it has been shown that such tests fail to detect a high percentage of TSO faults (see Section 2.2.4 as well as [60]). Additionally, being a charge-based fault model, a test might be invalidated by physical effects like glitches [61], [62] or charge-sharing [63], [64].

The generation of test patterns for TSOFs that are not invalidated by glitches has been studied extensively. The approaches that have been presented are generally based on one of two different ideas. The first group of approaches proposes circuit design methods that make invalidations impossible [65]–[68]. This requires the addition of extra logic to almost every cell of the circuit, increasing the overall circuit complexity. The result is an increased size and potentially a reduced speed. Similarly, re-designing the CMOS cells themselves [62], [63] can achieve the same result but suffers from the same problems. The second group of approaches focuses on generating test patterns that do not cause any glitches at the fault site – so called robust tests [69], [70]. While these tests are guaranteed to be valid, the analysis is pessimistic and patterns for some faults that could actually be safely tested are not found.

The challenge of glitches is not only relevant for the TSOF model but has also been considered for the test and diagnosis of transition-delay faults [71]–[73]. However, similar to the previously discussed methods for TSOFs, these approaches are also not based on an accurate consideration of the circuit's timing.

This chapter introduces a deterministic and accurate test pattern generation approach for TSOFs that does not require any modifications to the circuit but is also not based on pessimistic assumptions about the propagation of glitches. Instead, the power of SAT-based ATPG is leveraged to generate a high quality test set. Furthermore, the ATPG is capable of generating tests that utilize glitches to test faults which are considered to be untestable because of structural restrictions.

The chapter is structured as follows: Section 4.1 introduces a basic timing-unaware TSOF ATPG. The challenges that arise in TSOF testing (and that are not considered by the previously implemented basic ATPG) are discussed in Section 4.2. Thereafter, Section 4.3 shows how to automatically compute the detection patterns and characterize each of them with regard to the challenges based solely on the cell library. Section 4.4 focuses on the handling of glitches in SAT-based ATPG. The subsequent Section 4.5 discusses the implementation of the charge-sharing mitigation techniques. In Section 4.6 the glitch-based methods are further improved to make them more robust against minor changes in the circuit's timing. Next, Section 4.7 introduces the hazard and charge-sharing aware ATPG algorithm in detail. This algorithm is evaluated extensively in Section 4.8. Section 4.9 concludes the chapter with a summary.

This chapter is partially based on:

[J1] J. Burchard, D. Erb, S. M. Reddy, A. D. Singh, and B. Becker, "On the generation of waveform-accurate hazard and charge-sharing aware tests for transistor stuck-off faults in CMOS logic circuits", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2017. DOI: `10.1109/TCAD.2017.2772825`

[C2] J. Burchard, D. Erb, A. D. Singh, S. M. Reddy, and B. Becker, "Fast and waveform-accurate hazard-aware SAT-based TSOF ATPG", in *Design, Automation and Test in Europe (DATE), 2017*, Best Paper Award in the Test Category, 2017. DOI: `10.23919/DATE.2017.7927027`

[C3] J. Burchard, D. Erb, S. M. Reddy, A. D. Singh, and B. Becker, "Efficient SAT-based generation of hazard-activated TSOF tests", in *IEEE 35th VLSI Test Symposium (VTS)*, 2017. DOI: `10.1109/VTS.2017.7928943`

The main contributions by the author to this chapter are:

- The development of an automatic detection pattern generation tool that characterizes a cell library.
- The development and integration of an automatic detection of stability and charge-sharing requirements in the characterization tool.
- The development of an accurate SAT-based TSOF ATPG algorithm with an efficient hybrid formula encoding.
- The development and integration of advanced glitch-avoidance stability conditions into the ATPG.
- The development and integration of glitch-initialization techniques into this ATPG.
- The development and integration of charge-sharing mitigation techniques into this ATPG.
- The development of a fast preprocessing step to identify faults where timing information is not necessary to guarantee a successful test.
- The extensive analysis of the ATPG algorithm on a wide range of different circuits.

The implementations are built on top of the PHAETON framework [44] by Matthias Sauer which, among others, provides a circuit and Standard Delay Format (SDF) file parser, logic generators for the Tseitin transformation, the waveform accurate timing modeling as well as an interface to the SAT solver antom [13], [14].

4.1. Basic Transistor Stuck-Open Fault ATPG

The process of creating test patterns for transistor stuck-open faults with a SAT-based ATPG algorithm is in many ways similar to that of the SAT-based stuck-at fault ATPG that was introduced in the previous chapter. Nonetheless, there are certain differences which need to be considered. This section introduces the basic methodology of creating test patterns for TSOFs which will be extended throughout the chapter to handle all of the different challenges. The basic algorithm will then be known as the timing unaware ATPG for the remainder of this chapter.

4.1.1. Modeling Two Time Frames

To test for the presence of a TSOF reliably, a two pattern test is required. Thus, the ATPG needs to consider two time frames. As previously discussed, the circuit is unrolled once to create a separate copy for the first and second time frame. Just like in the stuck-at ATPG the analysis of the cones of influence is essential to produce a smaller formula. The different necessary parts for a TSOF are shown in Figure 4.1.

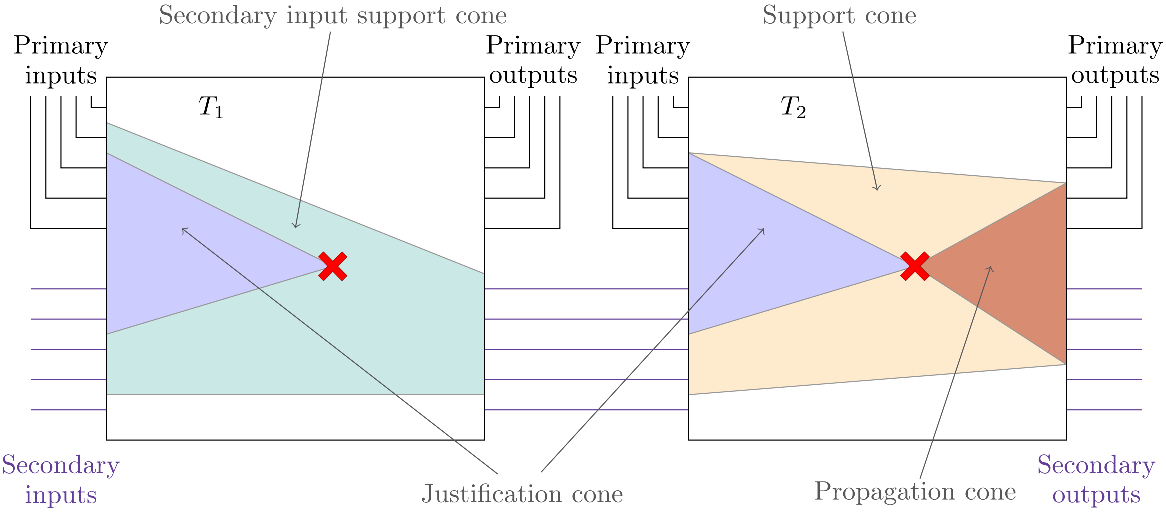


Figure 4.1.: Only the marked areas of the circuit in the two time frames are required when modeling the TSOF marked by the red cross.

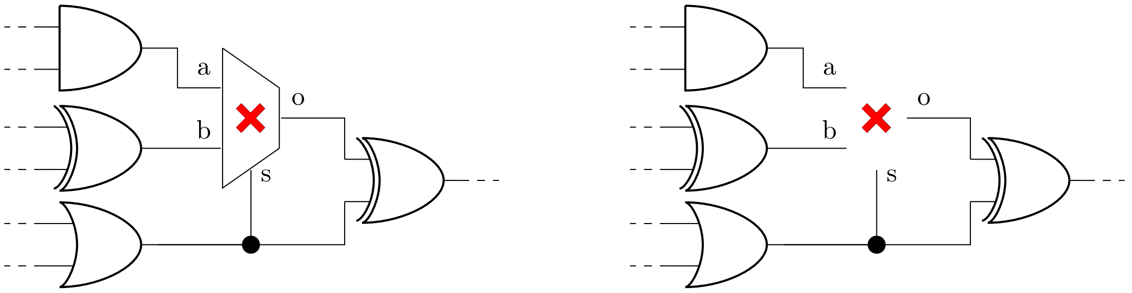
In the second time frame, the picture is similar to that for a stuck-at fault: The justification of the fault, the propagation of its effect and the support for the side inputs of the cells in the propagation cone are required. In the first time frame, on the other hand, only the justification cone of the fault is modeled because the propagation of the cell output is not relevant. Furthermore, a LOC type test requires that the flip-flop values that are used in the second time frame are supported in the first time frame. Hence, the cells in the secondary input support cone also have to be modeled. The faulty cell might be in this support cone (as shown in the picture). This is, however, not necessarily the case and irrelevant for the modeling. In the first time frame the initialization pattern is guaranteed to produce a correct output at the faulty cell's output – it behaves just like any other cell.

The cells in the marked areas can then be converted into a mapped circuit and transformed into a Boolean formula in CNF through the Tseitin transformation. The cells in the fault propagation cone are transformed twice – once to represent the propagation in the fault-free (good) circuit and once for that in the fault-affected (bad) one.

4.1.2. Modeling the Fault

The SAT-based stuck-at fault ATPG that was presented in Chapter 3 models the fault by splitting the faulty line into two parts. In the formula, the first part was forced to have the required activation value (e.g., '1' for a stuck-at-0 fault) whereas the second part had to have the correct value ('1') in the good circuit and the faulty value ('0') in the bad circuit (see Section 2.2.6).

The approach of modeling a TSOF is similar. However, unlike the stuck-at fault model which considers an input or output of a cell to be stuck, the TSOF model considers part of a cell itself to be faulty. Thus, instead of splitting the faulty line, the proposed ATPG algorithm removes the entire fault-affected cell from the circuit (see Figure 4.2).



(a) A circuit with a TSOF.

(b) The same circuit as modeled by the ATPG.

Figure 4.2.: Modeling a TSOF in a circuit by removing the faulty cell. The cell's inputs and output are then forced to specific values by the ATPG.

The variables representing the inputs and outputs of the faulty cell still occur in the final formula. However, the functionality of the faulty cell is not modeled. This task is taken over by the detection pattern which contains the conditions to activate the fault effect (i.e. the values that have to be present on the lines).

Assume that the TSOF in the multiplexer in Figure 4.2 can be detected by the detection pattern $\langle 000, 010 \rangle$, which causes the output to be '0' in the first time frame and '1' in the fault-free case in the second time frame. This detection pattern is encoded into the Boolean formula by adding the assignments shown in Figure 4.3. The indices of the variable indicate the time frame of the corresponding signal. In the second time frame, the output is modeled twice, once for the good and once for the bad circuit. The assignments can then be added as unit clauses to the formula, immediately forcing the corresponding variables to their required values.

The last step in the basic formula generation is the addition of the difference-detecting XOR-cells and the difference-enforcing clause that combines all of the outputs of the new cells. This step is the same as in the stuck-at ATPG as it is independent from the fault model. Furthermore, the difference between the good and the bad circuit in the

$$\begin{aligned} a_1 &= false & a_2 &= false \\ b_1 &= false & b_2 &= true \\ s_1 &= false & s_2 &= false \\ o_1 &= false & o_{2,good} &= true \\ & & o_{2,bad} &= false \end{aligned}$$

Figure 4.3.: List of the conditions that enforce the fault activation and set the output values for the considered TSOF in the multiplexer.

fault propagation cone can be modeled with the help of D-chains again. Based on the observations in Chapter 3, the hybrid D-chain with the dynamic node selection heuristic is utilized as it provides the largest average speedup across all of the benchmarks.

When the variables that correspond to the inputs and output of the – now removed – faulty cell are forced to the values required by the detection pattern, it is ensured that any satisfying assignment to the formula will correspond to a valid test pattern. This test pattern will apply the detection pattern to the fault site and create a difference between the faulty and the fault-free circuit on at least one of the observable circuit outputs.

4.1.3. Incremental Solving

In chapter 3 the evaluation of the stuck-at ATPG clearly showed that incremental solving can be very powerful because the solver can re-use some of its previous knowledge. The presented incremental approach appeared to be especially successful for the easy-to-detect faults which can be propagated to many outputs.

In the TSOF ATPG, the testability of the faults will probably be much lower because more difficult conditions (e.g., LOC restraints) need to be satisfied and not one but two time frames have to be considered. Thus, another incremental solving approach was selected.

The previous section described how the fault is modeled by removing the faulty cell from the circuit altogether and, instead, encoding the activation condition and output value of the cell directly as unit clauses into the formula. For an incremental approach the unit clauses are replaced by assumptions which can be removed again. Since the difference between different TSOFs in the same cell (from the ATPG perspective) lies only in the detection pattern, this allows for the re-use of the formula encoding a particular TSOF. The ATPG flow is shown in Figure 4.4. The entire formula is only built up once per cell. Patterns for the different TSOFs in the cell are then generated by incrementally adding the different detection and propagation properties as assumptions.

Furthermore, some faults might be detectable by different detection patterns. Here, again, the same basic formula can be re-used. If a test pattern for a particular fault is detected with a detection pattern, all other detection patterns for the same fault are skipped.

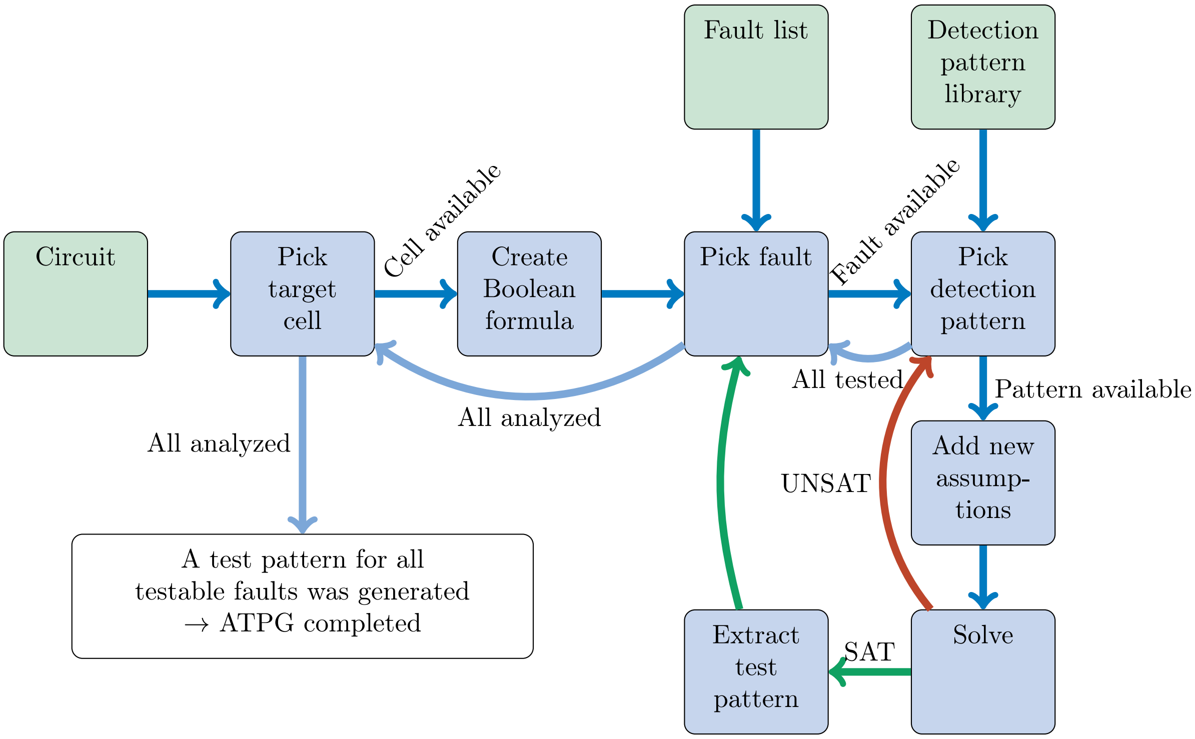


Figure 4.4.: Overview of the basic SAT-based TSOF ATPG flow. The Boolean formula is only created once per cell. The different faults and different detection patterns per fault are then added as assumptions to this formula.

Overall, the proposed incremental approach reduces the formula generation time per fault significantly while still ensuring that hard-to-detect or untestable faults can be detected quickly without having to solve a huge number of small incrementally built up formulas.

4.2. Challenges

After the previous section, it might appear that generating test patterns for transistor stuck-open faults differs little from the generation of test patterns for other fault models. However, testing a TSOF is challenging because of three effects: Firstly, a glitch might invalidate an otherwise fine test pattern and mask the effect of a fault. Similarly, a charge-sharing conflict could have the same effect. Furthermore, applying a detection pattern to the faulty cell might not be possible because of test infrastructure restrictions like LOC or LOS. This section discusses the reasons for each of these challenges and gives an example for each of them. The subsequent sections highlight the novel approaches of overcoming them that are integrated into the presented ATPG algorithm.

It should be noted that the discussed effects are not restricted to the TSOF model, but could arise in similar low-level fault models as well. Most notably, the cell-aware fault model [33] also considers cell internal opens and might be faced with the same kind of challenges that could be solved similar to those presented here.

4.2.1. Glitches

The propagation of a signal change at the input of a cell to its output requires a certain amount of time. In a circuit which consists of many cells, these delays add up along the paths through the circuit and can cause signal changes to arrive at a cell at different times. This, in turn, might result in glitches at the output of the cell. As an example, consider Figure 4.5 which shows the propagation of the input assignment $\langle 11, 00\rangle$ through a circuit. The blue waveforms correspond to the signal propagation. Relevant time points are annotated with the discrete time step at which the change occurs. Due to the different delays, there is a glitch at the output of the XOR-cell.

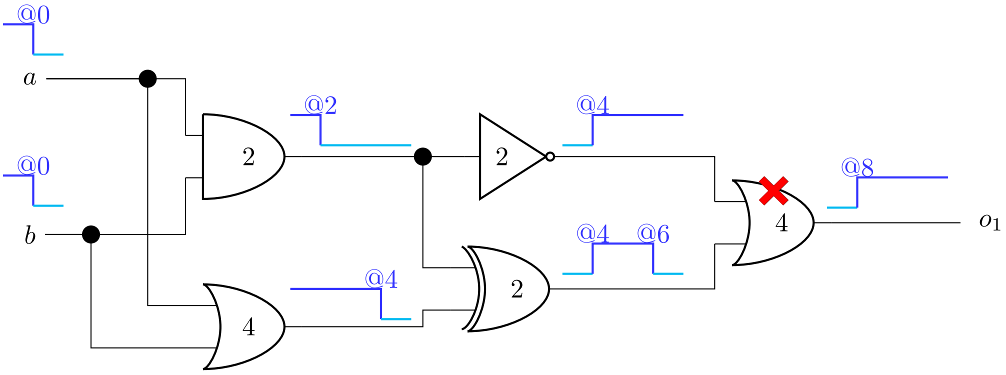


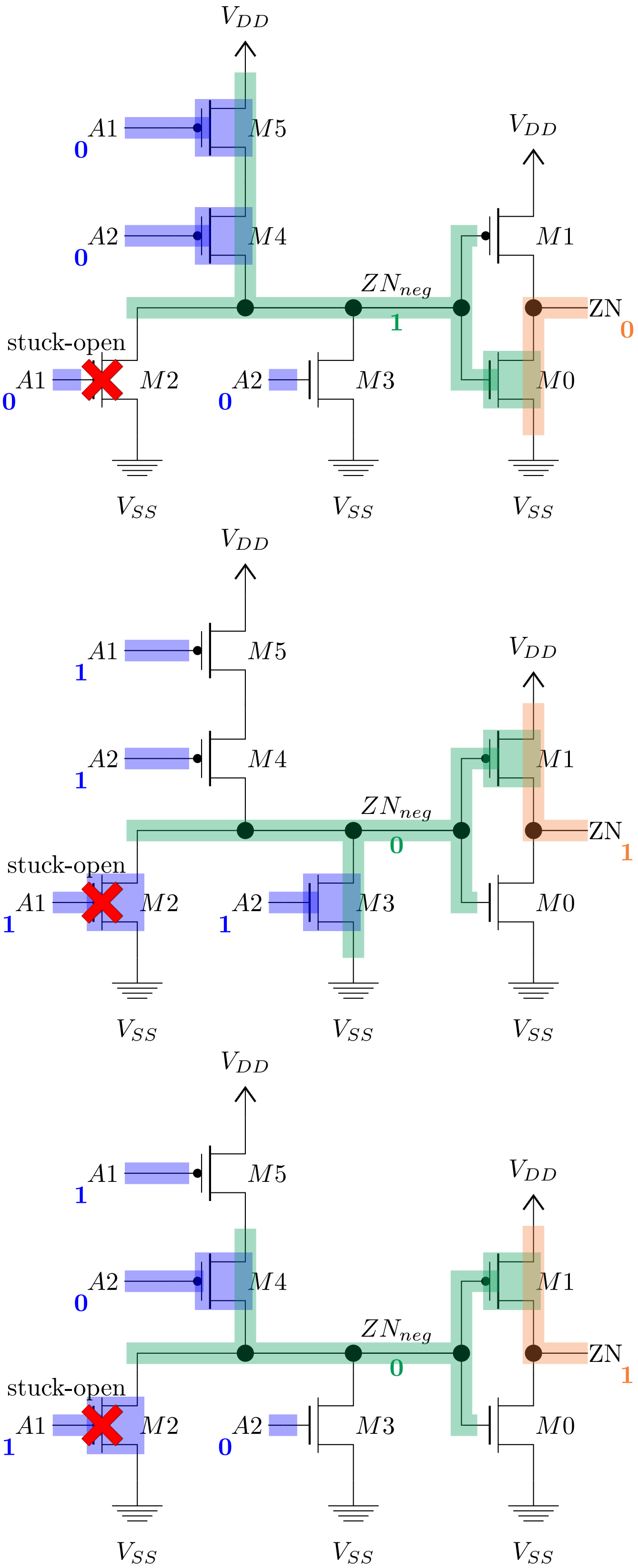
Figure 4.5.: Propagation of signal changes through a circuit. The value in each cell indicates the delay from the inputs to the output.

Assume that the right OR-cell in the circuit in Figure 4.5 (marked with a red cross) has a TSOF in transistor $M2$. As previously discussed in Section 2.2.4, applying the detection pattern $\langle 00, 10\rangle$ to the OR-cell makes the fault effect visible at its output (cf. Figure 2.11 in Chapter 2).

In a classical two time frame modeling of the circuit, the glitch at the output of the XOR-cell would not be modeled and it would be assumed that the detection pattern is successfully applied to the faulty cell (since its inputs are "00" in the first time frame and "10" after all the changes have stabilized). In reality, three different patterns are applied to the faulty cell: First "00", then "11" (from time step 4 to 6 because of the glitch) and, finally, "10". Figure 4.6 shows the behavior of the transistors and the stored values in the cell over the three input assignments.

To detect the TSOF in $M2$, the cell output must stay at '0' after applying $\langle 00, 10\rangle$. Instead, the glitch on the second input $A2$ charges the output to '1' through $M3$ (which is positioned parallel to $M2$). The result is that the fault effect is masked – the cell appears to be working correctly although it actually has a TSOF.

This example highlights the importance of considering glitches when testing for TSOFs. As the conducted experiments will show, test patterns generated without glitch-awareness have a large probability of failing to detect the considered fault which could result in a significant DPPM increase. More details will be given in Section 4.4 which discusses the handling of glitches by the presented ATPG.



The pattern **“00”** turns the transistors $M5$ and $M4$ on (blue).
This creates a conducting path from V_{DD} to the gate input of $M0$ and turns it on (green).
By turning on $M0$, the cell output ZN is connected to V_{SS} and, hence, becomes ‘0’ (orange).

The pattern **“11”** turns on the transistors $M2$ and $M3$ (blue). However, because of the TSOF $M2$ stays off. Nonetheless, a conducting path from V_{SS} to the gate input of $M1$ still exists through $M3$ (green).
By turning on $M1$, the cell output ZN is connected to V_{DD} and, hence, becomes ‘1’ (orange).

The pattern **“10”** turns on the transistors $M4$ and $M2$ (blue). However, because of the TSOF $M2$ stays off. Thus, the internal signal ZN_{neg} is connected to neither V_{DD} nor V_{SS}. The signal is floating and retaining its previous value (‘0’). Therefore, $M1$ stays active, the cell output ZN is still connected to V_{DD} and stays ‘1’ (orange).

Figure 4.6.: Applying “00”, then “11” and, finally, “10” to an OR-cell with a TSOF in $M2$.

4.2.2. Charge-Sharing

The second challenge when testing TSOFs is charge-sharing. Charge-sharing occurs when the propagation pattern connects intra-cell lines with an opposite charge to the floating line. The outcome is a charge that corresponds to a voltage that is neither clearly '1' nor '0'. Depending on the interpretation of the voltage by the subsequent transistors the fault effect might be masked.

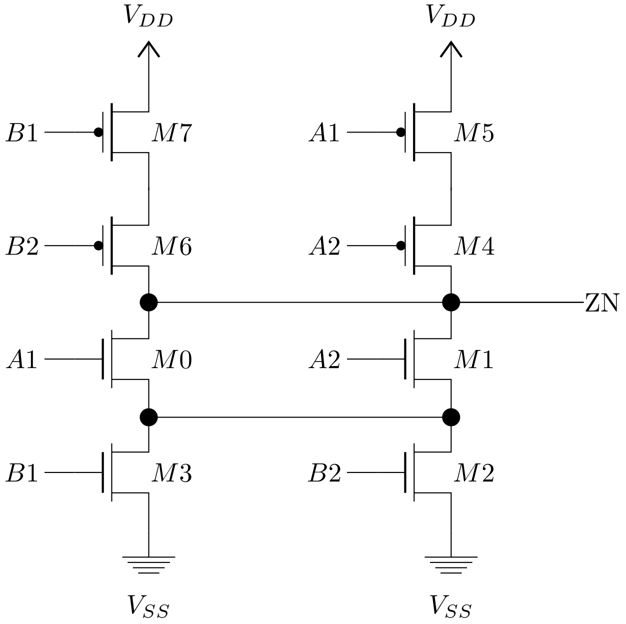


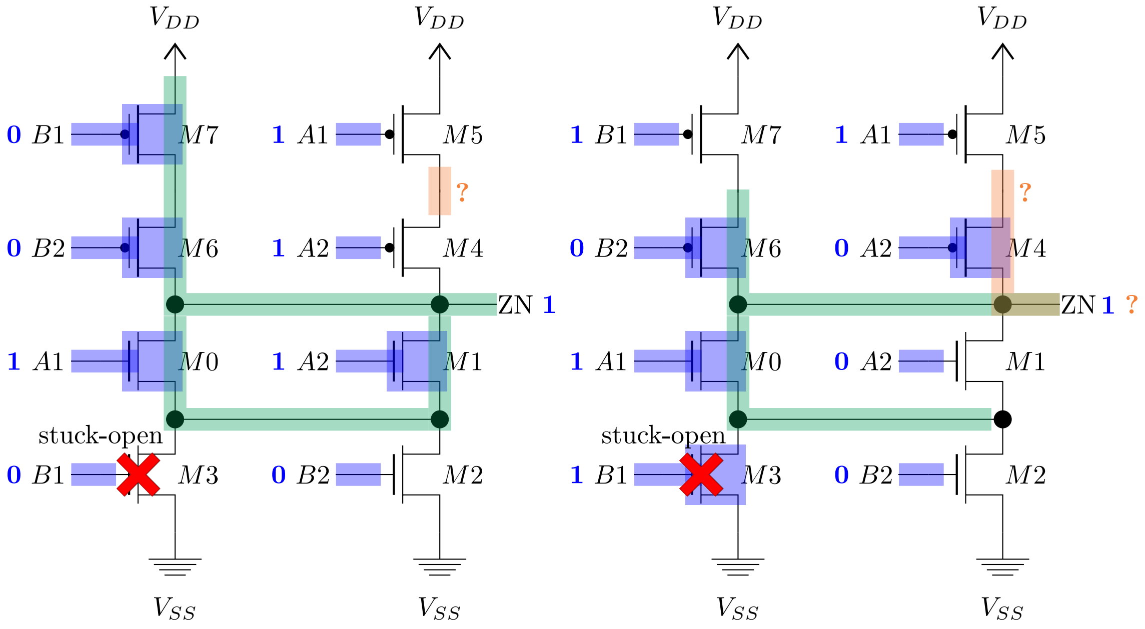
Figure 4.7.: Transistor layout of an OR-AND-Invert-cell.

As an example, consider the OR-AND-Invert-cell in Figure 4.7 with a TSOF in $M3$. The pattern $\langle 1100, 1010\rangle$ appears to be a valid detection pattern – "1100" connects the output to V_{DD}, while "1010" makes it float and retain its previous value instead of switching to '0' (see Figure 4.8).

However, in the first time frame there is another line that is floating: the interconnect between $M4$ and $M5$ (shown in orange). In the second time frame, this line is then connected to the cell's output through $M4$ and the charges of the two floating lines are shared. If the $M4$-$M5$ interconnect has a charge equivalent to '0', the '1' charge of ZN is reduced because some of it is transfered to the interconnect. Whether the corresponding voltage on ZN is still interpreted as '1' depends on the remaining charge which, in turn, depends on the capacitances of the different interconnects and transistors.

While in this example the number of nodes that have a charge of '1' is rather large and the test would most likely be successful[1], avoiding the conflict (and the chance of test invalidation) altogether is still desirable. This could, for example, be achieved through another detection pattern: By applying "1000" to the cell in the first time frame the $M4$-$M5$ interconnect would definitely be charged to '1' and no charge-sharing conflict could arise.

[1] In [67], the authors even noted that charge-sharing conflicts are generally improbable.



(a) Applying “1100” to the cell.

(b) Applying “1010” to the cell.

Figure 4.8.: The problem of charge-sharing in TSOF-testing: In the second time frame, the charge of ZN is shared with an unknown charge from the $M4$-$M5$ interconnect.

Section 4.3 discusses the generation of detection patterns that are guaranteed to be without a charge-sharing conflict or for which the charge-sharing conflict can be mitigated.

4.2.3. Test Infrastructure Restrictions

The final challenge in TSOF testing stems from the test infrastructure itself. Generally, the flip-flop values for the second time frame cannot be chosen freely due to test infrastructure and cost constraints. Instead, methods like LOC and LOS are used. Here, the values of the flip-flops in the second time frame originate from their inputs in the first time frame. While this gives a tremendous test time reduction, it also restricts the input values of the second time frame. In the worst case, some faults cannot be tested at all because their detection pattern cannot be justified through the first time frame.

However, it might be possible to apply the required initialization pattern through glitches while switching from T_1 to T_2. In this case, the pattern T_1 is used to justify the flip-flop values that are required in the second time frame but does not initialize the fault. The initialization only occurs while switching from T_1 to T_2. Such a glitch-initialization of the fault allows for the testing of faults which would otherwise be considered as untestable. This will be analyzed further in Section 4.4.

4.3. Detection Library

To test for the existence of a TSOF, a special detection pattern has to be applied to the inputs of the faulty cell. This detection pattern must satisfy certain properties and might have special requirements.

The basic property of a detection pattern is that it makes the fault effect visible at the cell's output. For some TSOFs, this might only be possible if some of the inputs are without glitches (stable). Furthermore, the detection pattern must also ensure that no charge-sharing conflict arises or that the conflict is mitigated.

The possible detection patterns for a fault are computed in a preprocessing step that needs to be performed only once for each cell in a cell library. The overall flow in creating a list of detection patterns for a single cell is shown in Figure 4.9.

The analysis is based on a transistor level description of the cell in question that is provided with the cell library. This file contains a textual representation of the transistor circuits shown throughout the chapter in the spice format. The detection pattern computation itself works as a very basic ATPG algorithm that is based on simulation and can be split into the following four steps:

1. Simulation of all possible $\langle T_1, T_2 \rangle$ patterns in a fault-free and faulty version of the cell. Patterns that produce a difference between the two versions at the output of the cell are stored as detection pattern candidates.

2. Simulation of all possible switching orders for each detection pattern candidate. This step ensures that no matter in which order the signal transitions arrive at the faulty cell, the detection pattern will always detect the fault. In the considered example, the detection pattern candidate $\langle 1010, 0100 \rangle$ is dropped because there is a switching order that hides the fault effect: 1010 → **0**010 → 00**0**0 → 0**1**00. The intermediate pattern "0000" opens a path in parallel to the faulty $M7$ (through $M4$ and $M5$) that charges the output to '1', masking the fault effect.

3. For each detection pattern candidate, check if it can be invalidated by glitches. To this end, the switching from T_1 to T_2 is simulated with additional glitches on all of the not switching inputs. Again, all possible switching orders are considered. Thus, glitches of any length and time of occurrence are covered. As an example, consider the detection pattern candidate $\langle 1111, 1100 \rangle$ where both $A1$ and $A2$ do not switch. One possible switching order with a glitch on both $A1$ and $A2$ is: 1111 → 0111 → 0011 → 00**0**1 → 000**0** → 0100 → 1100 (switches due to a glitch are highlighted in red). As shown in the previous step, the intermediate pattern "0000" hides the fault effect. Thus, a glitch at both $A1$ and $A2$ must be avoided to ensure a successful test of the fault. However, it is sufficient if one of the two inputs is glitch free (also referred to as stable) to ensure a successful test. Any stability requirement is added to the detection pattern candidate.

4. The final analysis step checks if there is a charge-sharing conflict for the detection pattern candidate. To this end, all possible switching orders are simulated again. While propagating the induced changes through the cell, it is checked whether the charge of the floating signal line is connected to any other floating line that was not

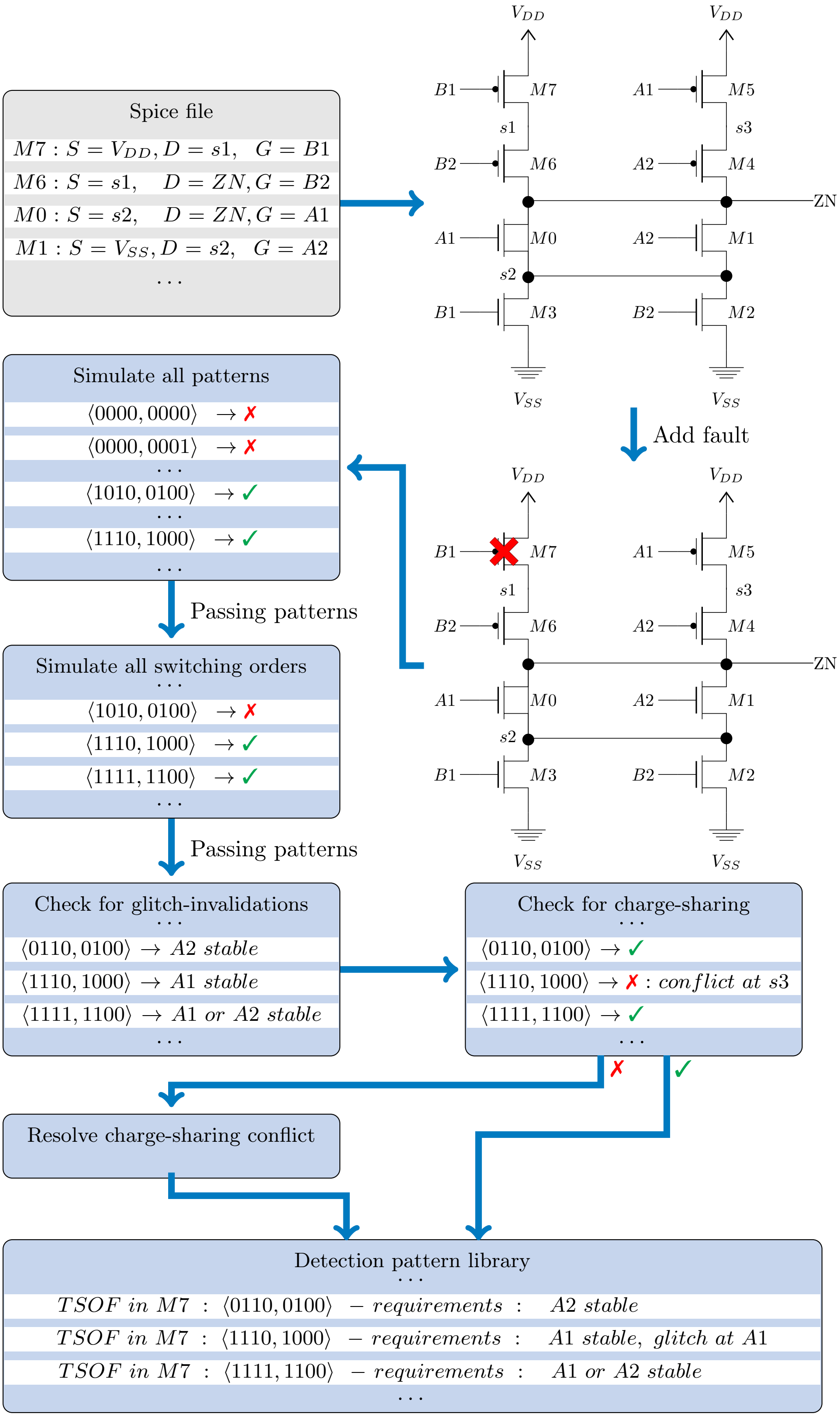

Spice file
$M7 : S = V_{DD}, D = s1, \quad G = B1$
$M6 : S = s1, \quad D = ZN, G = B2$
$M0 : S = s2, \quad D = ZN, G = A1$
$M1 : S = V_{SS}, D = s2, \quad G = A2$
...
V_{DD}
V_{DD}
$B1$
$M7$
$A1$
$M5$
$s1$
$s3$
$B2$
$M6$
$A2$
$M4$
ZN
$A1$
$M0$
$A2$
$M1$
$s2$
$B1$
$M3$
$B2$
$M2$
V_{SS}
V_{SS}
Add fault
Simulate all patterns
$\langle 0000, 0000 \rangle \rightarrow$ ✗
$\langle 0000, 0001 \rangle \rightarrow$ ✗
...
$\langle 1010, 0100 \rangle \rightarrow$ ✓
...
$\langle 1110, 1000 \rangle \rightarrow$ ✓
...
Passing patterns
Simulate all switching orders
...
$\langle 1010, 0100 \rangle \rightarrow$ ✗
$\langle 1110, 1000 \rangle \rightarrow$ ✓
$\langle 1111, 1100 \rangle \rightarrow$ ✓
...
Passing patterns
Check for glitch-invalidations
...
$\langle 0110, 0100 \rangle \rightarrow A2\ stable$
$\langle 1110, 1000 \rangle \rightarrow A1\ stable$
$\langle 1111, 1100 \rangle \rightarrow A1\ or\ A2\ stable$
...
Check for charge-sharing
...
$\langle 0110, 0100 \rangle \rightarrow$ ✓
$\langle 1110, 1000 \rangle \rightarrow$ ✗ $: conflict\ at\ s3$
$\langle 1111, 1100 \rangle \rightarrow$ ✓
...
✗
✓
Resolve charge-sharing conflict
Detection pattern library
...
$TSOF\ in\ M7 : \langle 0110, 0100 \rangle \ - requirements : \ A2\ stable$
$TSOF\ in\ M7 : \langle 1110, 1000 \rangle \ - requirements : \ A1\ stable,\ glitch\ at\ A1$
$TSOF\ in\ M7 : \langle 1111, 1100 \rangle \ - requirements : \ A1\ or\ A2\ stable$
...

Figure 4.9.: Overview of the developed detection library computation flow.

charged to the right value in T_1 or not charged at all. If so, the pattern is marked and the charge-sharing conflict has to be resolved in an extra step which is discussed below. Otherwise, the detection pattern candidate is added to the detection pattern library directly.

Overall, the computation of the detection library depends on different simulations to compute valid detection patterns, to test their vulnerability to glitches and to check for any charge-sharing conflicts. This simulation-heavy approach is feasible because even the most complex cells in the considered cell libraries have only six inputs and the total number of different possible input assignments is, therefore, very low.

If a cell library contains very complex cells with a large number of inputs, the simulation could also be replaced by a transistor level ATPG algorithm for detection patterns. Since the utilized cell library does not contain any such cells, this was not pursued further in this work.

4.3.1. Resolving Charge-Sharing Conflicts

When a detection pattern is susceptible to charge-sharing, this conflict can be resolved by different means. These aim at charging the conflicting line (referred to as *cl* in this section) to the correct value before it is connected with the line that is storing the relevant charge between T_1 and T_2. In this work, three different techniques for the resolution of charge-sharing conflicts are considered and implemented:

1. Charge the *cl* by ensuring a certain switching order.
2. Charge the *cl* by a glitch before switching from T_1 to T_2.
3. Charge the *cl* in an additional time frame T_0.

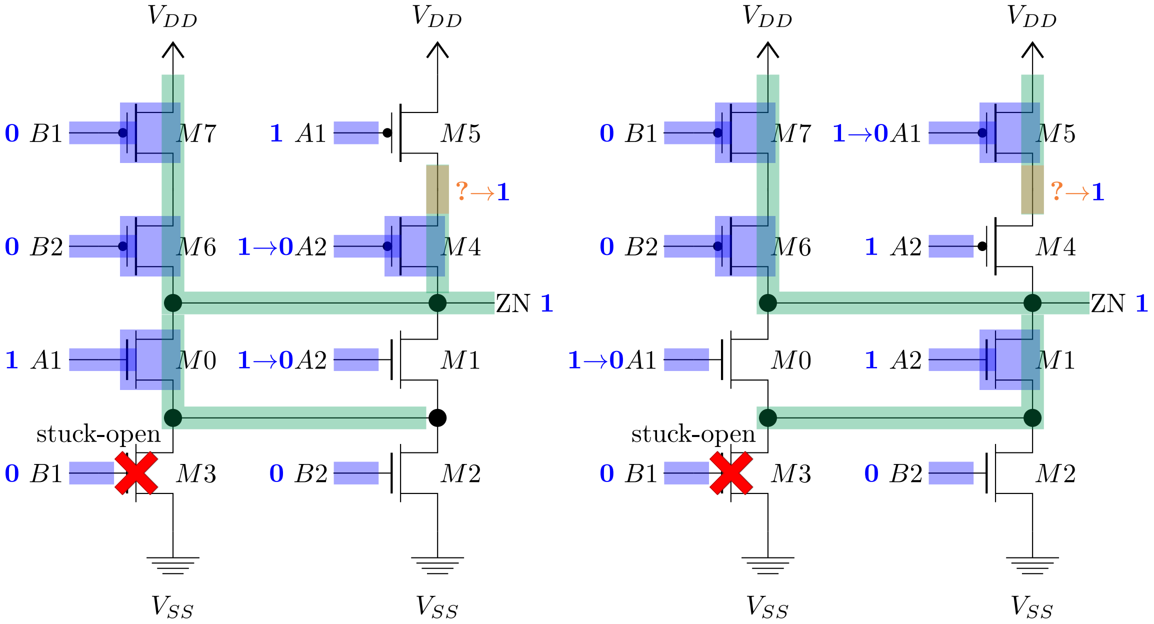
The first two techniques can be considered as additional requirements for the detection pattern itself and are referred to as mitigation techniques, while the third technique requires an entire additional time frame.

Mitigating the Conflict

The idea of the mitigation techniques is to utilize the switching activity of the circuit to charge the *cl*. As an example, consider the detection pattern $\langle 1100, 1010\rangle$ for a TSOF in transistor $M3$ of an OR-AND-Invert-cell as discussed in Section 4.2.2 again. Here, the $M4$-$M5$ interconnect is the *cl* with an unknown charge. There are two ways of charging the *cl*: Either the input $A2$ switches from ‘1’ to ‘0’ first. This opens $M4$ and charges the *cl* to the required value, ‘1’ (see Figure 4.10a). Alternatively, a glitch on the input $A1$ opens $M5$ and has the same effect (see Figure 4.10b).

If a mitigation technique was found to work for a detection pattern that suffers from a charge-sharing conflict, it is added as a requirement to the pattern. For example, in Figure 4.9 the pattern $\langle 1110, 1000\rangle$ has the requirement of a glitch at input $A1$ to mitigate the charge-sharing conflict and, furthermore, a stability requirement at input $A1$. This combination can indeed occur and lead to a valid test but necessitates the combination of glitch-initialization and the weak stability condition that will be introduced in Section 4.4.1. The implementation of the mitigation techniques themselves will be discussed in Section 4.5.

To simplify the modeling of the mitigation conditions in the SAT formula, only permanent charge-sharing resolutions – where the conflicting line cannot be discharged through a glitch again – are added to the detection library.



(a) By switching $A2$ first.

(b) Through a glitch on $A1$.

Figure 4.10.: Mitigating the charge-sharing conflict by charging the $M4$-$M5$ interconnect.

Adding a Pre-Charging Time Frame

A pre-charging time frame can provide another option to charge a *cl* to the required value. Here, the two pattern test $\langle T_1, T_2 \rangle$ is extended into a three pattern test $\langle T_0, T_1, T_2 \rangle$ where T_0 must charge the *cl*. Considering a three pattern test during the test pattern generation greatly increases the difficulty of the problem. Thus, the approach only considers pre-charging patterns that reliably charge the *cl* independent of the switching order from T_0 to T_1 or any occurring glitches.

To find the pre-charging patterns for a detection pattern every input combination is tested for its validity as a T_0 pattern. Again, all possible switching orders with all possible glitches are simulated for each T_0 candidate. Only if the *cl* is charged for any such possibility, the pattern is added as a pre-charging pattern.

The implementation the of pre-charging time frame will be discussed in combination with the overall ATPG flow in Section 4.7.

4.4. Handling Glitches

Recall that a detection pattern defines the input assignment that is required at the inputs of the fault-affected cell to make the fault effect visible whereas a test pattern is applied to the circuit inputs. While the detection library defines the requirements of each detection pattern, these requirements still have to be fulfilled by the test patterns generated by the ATPG algorithm. The main challenge lies with computing test patterns that satisfy the stability conditions of a detection pattern. This part demands advanced modeling techniques and explains why no other TSOF ATPG supports glitch-aware test pattern generation.

In this section and the subsequent subsections the newly developed techniques which allow the presented TSOF ATPG to generate valid test patterns are described. The effort can be grouped into two areas: glitch-avoidance and glitch-initialization. Glitch-avoidance focuses on ensuring that glitches do not occur when they might be harmful, whereas glitch-initialization attempts to utilize glitches to mitigate charge-sharing conflicts or to circumvent test infrastructure restrictions.

All of the presented techniques are based on the waveform accurate circuit model introduced by [48] and discussed in Section 2.2.7. Thus, for every modeled signal there is a list $Tvars$, containing one variable for each time step in which the signal value might change. The earliest time at which a signal can change is t_{first}. Accordingly, the latest time is t_{last}. For a nicer representation it is assumed that the $Tvars$ of the inputs of the cell always have the same t_{first} and the same length. In reality, mismatching $Tvars$ can be adapted by repeating the first and/or last value (since there are no changes before or after the $Tvar$ list) until all $Tvars$ are matched. For the handling of the glitches, it can be assumed that the entire circuit is modeled with waveform accuracy. Exact details on the overall construction of the ATPG will be given in Section 4.7.

4.4.1. Glitch-Avoidance

A test pattern can have stability requirements for some of the inputs which do not change from T_1 to T_2. The goal is to encode these requirements into the formula used by the SAT-based ATPG algorithm which ensures that the requirements are satisfied by any generated pattern automatically. To this end, the formula Φ created by the basic, timing-unaware TSOF ATPG algorithm presented in Section 4.1 is extended with stability conditions.

In the ATPG, stability can be enforced in two different manners: Through the cheap but more pessimistic strong stability condition or through the more complex but also more exact weak stability condition. The latter also requires a more complex modeling of the fault activation condition which is discussed in combination with glitch-initialization in the next section.

Strong Stability Condition

The strong stability condition enforces strict glitch freedom for any input i with a stability requirement. This means that i must stay at its T_1 value without any transitions during the propagation of the changes induced by applying T_2 to the circuit. Transferred to the

Boolean formula that models the signal propagation, this implies that all of the $Tvars_i$ from t_{first} until t_{last} must have the same value.

Since Φ is utilized incrementally for different faults and detection patterns, each of which might have different stability requirements, the strong stability condition is encoded with an extra variable *stableStrong*$_i$ which controls whether stability is enforced at input i. When *stableStrong*$_i$ is set to *true* by an assumption, any solution to the Boolean formula corresponds to a test pattern that does not produce a glitch at cell input i. If *stableStrong*$_i$ is set to $false$, the stability condition can be ignored and all of the corresponding clauses are satisfied already.

The strong stability condition and the controlling variable can be efficiently encoded into the Boolean formula:

$$\bigwedge_{t=t_{first}}^{t_{last}} stableStrong_i \Rightarrow \Big(Tvars_i[t] \Rightarrow Tvars_i[(t+1) mod\ (t_{last}+1)]\Big) \tag{4.1}$$

Here, *mod* describes the modulus operation which gives the remainder after integer division. As Figure 4.11 shows, the implications form a circular dependency which ensures that all variables are assigned to the same value: Whenever any one of the $Tvars_i$ is assigned to *true*, all other $Tvars_i$ must also become *true*. Conversely, it is only possible to assign one of the $Tvars_i$ to $false$ when all other $Tvars_i$ are also assigned to $false$.

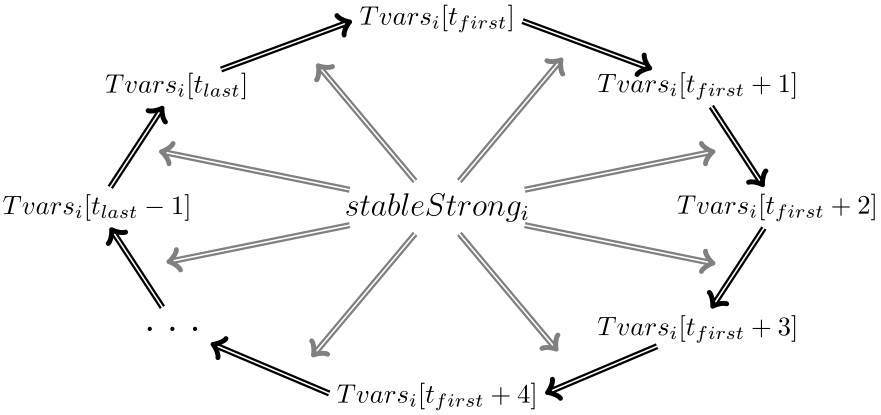


Figure 4.11.: The circular implication which ensures total glitch freedom at input i of the fault-affected cell when *stableStrong*$_i$ = *true*.

Formula 4.1 can be transformed into CNF easily because it consists only of implications. The final list of clauses that encode strong stability at input i is shown in Figure 4.12. When *stableStrong*$_i$ is assumed to be $false$, all of the clauses become satisfied right away and will be ignored by the solver for the remainder of the solve process. Thus, if a stability condition is not required for a particular detection pattern it only adds a minimum burden onto the SAT solver.

Weak Stability Condition

The weak stability condition is based on a more detailed look at the glitches that occur at a cell. Recall that glitches are a problem because they might charge the line that should

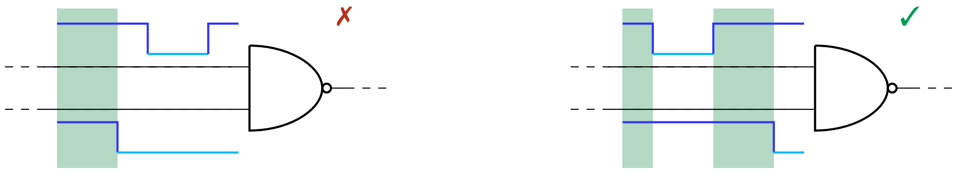
$$
\begin{aligned}
&(\neg stableStrong_i \vee \neg Tvars_i[t_{first}] &&\vee Tvars_i[t_{first}+1]) && (4.2)\\
&(\neg stableStrong_i \vee \neg Tvars_i[t_{first}+1] &&\vee Tvars_i[t_{first}+2]) && (4.3)\\
&(\neg stableStrong_i \vee \neg Tvars_i[t_{first}+2] &&\vee Tvars_i[t_{first}+3]) && (4.4)\\
&(\neg stableStrong_i \vee \neg Tvars_i[t_{first}+3] &&\vee Tvars_i[t_{first}+4]) && (4.5)\\
&\qquad\qquad\qquad \dots\\
&(\neg stableStrong_i \vee \neg Tvars_i[t_{last}-1] &&\vee Tvars_i[t_{last}]) && (4.6)\\
&(\neg stableStrong_i \vee \neg Tvars_i[t_{last}] &&\vee Tvars_i[t_{first}]) && (4.7)
\end{aligned}
$$

Figure 4.12.: List of the clauses that encode the strong stability condition.

be floating due to the TSOF to the value it would have in the correct circuit by opening a path in parallel to the faulty transistor – and, hence, masking the fault. Thus, if the line should stay at ‘0’ due to the fault, the glitch would charge it to ‘1’ and vice versa. The initial charging of the line to ‘0’ is the sole task of the initialization pattern in the first time frame. Hence, every time the initialization pattern is applied to the faulty cell, the internal line will be charged to the necessary value (again).

The weak stability condition uses this fact and allows some glitches to occur. It only enforces stability from the point in time onward where a glitch might actually permanently mask the fault effect. Every application of the initialization pattern re-charges the relevant internal line to the correct value. Thus, the weak stability condition allows for the occurrence of glitches before the last time at which the initialization pattern is applied to the cell. It should be noted that the weak stability condition is a replacement for the previously introduced strong stability condition.

As an example, consider a $NAND$-cell with a TSOF in transistor $M3$ which needs the detection pattern $\langle 11, 10\rangle$ and stability on the first input. Figure 4.13 shows two different signal traces at the inputs of the cell.



(a) The glitch occurs after the initialization and hides the fault effect.

(b) The glitch occurs before the last initialization and the test is valid.

Figure 4.13.: Advanced glitch timing considerations that are modeled by the weak stability condition.

Marked in green are the time points at which the initialization pattern is applied to the cell. In Figure 4.13a the glitch occurs after the initialization pattern and a successful test cannot be guaranteed. In Figure 4.13b, on the other hand, the initialization pattern is again applied to the cell after the glitch and the test is successful. While the strong stability condition would consider both of these examples as a violation of the input stability requirement, the weak stability condition allows for the case sketched in Figure 4.13b. Nonetheless, the weak stability condition still ensures that the case of Figure 4.13a does not occur.

To model if a glitch occurred before or after the last initialization a new variable vector $stableWeak_i$ of the same size as $Tvars_i$ is introduced. The variable $stableWeak_i[t]$ indicates if the input i will be stable from t onwards. If the variable at point t is assigned to $true$, then all subsequent variables at $t+1, t+2, \ldots$ must also be assigned to $true$. Furthermore, the signal may of course not change between any of the subsequent time steps. Because $stableWeak_i$ cannot simply consider all of the $Tvars_i$, the encoding of these properties as a Boolean formula is more complex:

$$stableWeak_i[t_{last}] = true \tag{4.8}$$

$$\bigwedge_{t=t_{first}}^{t_{last}-1} stableWeak_i[t] \Rightarrow \Big(stableWeak_i[t+1] \wedge \big(Tvars_i[t] \Leftrightarrow Tvars_i[t+1] \big) \Big) \tag{4.9}$$

The recursive definition starts at the last point of activity (after which the signal will definitely be stable) and works backwards until covering the entire $Tvar$ vector. Figure 4.14 shows the scope and implications for each single $stableWeak_i[t]$ variable.

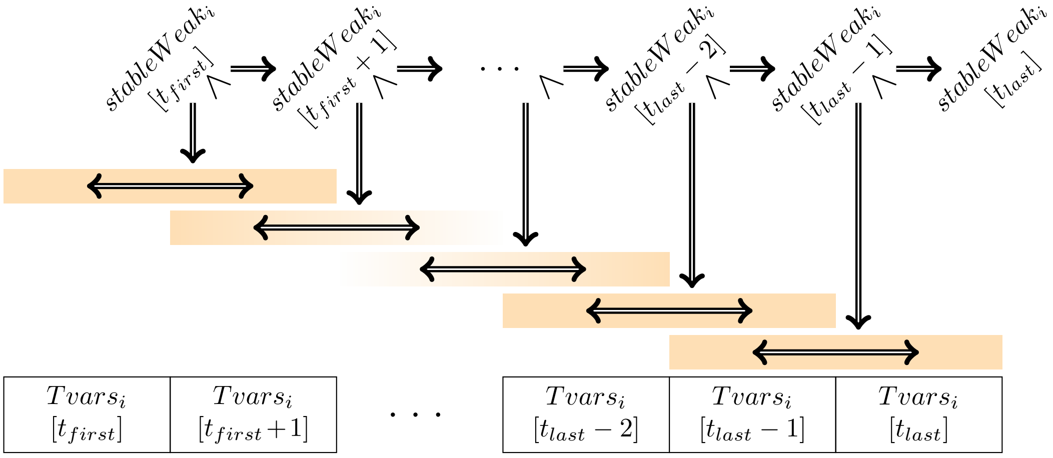

$stableWeak_i$ $[t_{first}]$
$stableWeak_i$ $[t_{first}+1]$
$\cdots$
$stableWeak_i$ $[t_{last}-2]$
$stableWeak_i$ $[t_{last}-1]$
$stableWeak_i$ $[t_{last}]$
$\wedge$
$Tvars_i$ $[t_{first}]$
$Tvars_i$ $[t_{first}+1]$
$\cdots$
$Tvars_i$ $[t_{last}-2]$
$Tvars_i$ $[t_{last}-1]$
$Tvars_i$ $[t_{last}]$

Figure 4.14.: The implications that are required to model the weak stability condition. To assign a $stableWeak_i$ variable to $true$, all outgoing implications need to be satisfied.

Due to its more complex structure, more clauses are necessary to model the weak stability condition as shown in Figure 4.15. For each time step at which the signal might be active, three clauses have to be added to the formula (compared to only one clause per time step for the strong stability condition).

Furthermore, by itself the vector $stableWeak_i$ is insufficient to ensure that there are no more glitches after the initialization pattern has been applied for the last time. For this task a second vector $init$, which stores when the fault is initialized, is indispensable. While the total cost to model the weak stability condition is higher than the cost for the

$$\big(\ \ stableWeak_i[t_{last}]\big) \tag{4.10}$$
$$\big(\neg stableWeak_i[t_{last}-1] \lor stableWeak_i[t_{last}]\big) \tag{4.11}$$
$$\big(\neg stableWeak_i[t_{last}-1] \lor \ \ Tvars_i[t_{last}-1] \lor \neg Tvars_i[t_{last}]\big) \tag{4.12}$$
$$\big(\neg stableWeak_i[t_{last}-1] \lor \neg Tvars_i[t_{last}-1] \lor \ \ Tvars_i[t_{last}]\big) \tag{4.13}$$
$$\big(\neg stableWeak_i[t_{last}-2] \lor stableWeak_i[t_{last}-1]\big) \tag{4.14}$$
$$\big(\neg stableWeak_i[t_{last}-2] \lor \ \ Tvars_i[t_{last}-2] \lor \neg Tvars_i[t_{last}-1]\big) \tag{4.15}$$
$$\big(\neg stableWeak_i[t_{last}-2] \lor \neg Tvars_i[t_{last}-2] \lor \ \ Tvars_i[t_{last}-1]\big) \tag{4.16}$$
$$\dots$$
$$\big(\neg stableWeak_i[t_{first}] \quad \lor stableWeak_i[t_{first}+1]\big) \tag{4.17}$$
$$\big(\neg stableWeak_i[t_{first}] \quad \lor \ \ Tvars_i[t_{first}] \lor \neg Tvars_i[t_{first}+1]\big) \tag{4.18}$$
$$\big(\neg stableWeak_i[t_{first}] \quad \lor \neg Tvars_i[t_{first}] \lor \ \ Tvars_i[t_{first}+1]\big) \tag{4.19}$$

Figure 4.15.: List of the clauses that encode the weak stability condition for input i.

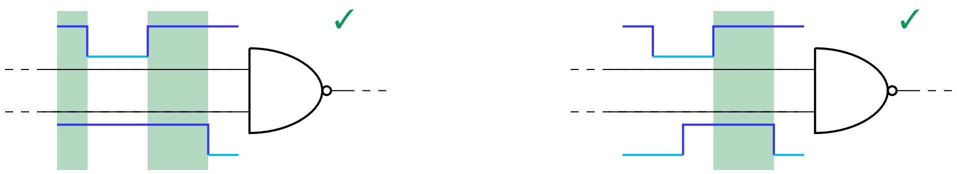
strong stability condition, *init* can also be used to model the glitch-initialization of the fault. Thus, the details of *init* are described in the next section.

4.4.2. Glitch-Initialization

When a fault cannot be initialized by the T_1 pattern, it might still be possible to initialize it through a glitch while switching from T_1 to T_2. To test a TSOF only two requirements are relevant:

1. The fault has to be initialized at some point in time.
2. When switching to the propagation pattern, the test must not be invalidated through a glitch.

These conditions are exactly what is encoded by the weak stability condition. Thus, when modeling the weak stability condition, glitch-initialized faults are also automatically considered and vice-versa. Figure 4.16 shows an example for a fault which can be tested because of considering weak stability as well as an example for a fault that can be tested because it is initialized by a glitch.



(a) Weak stability

(b) Glitch-initialization

Figure 4.16.: Comparison of the weak stability condition and glitch-initialization.

In both cases, only the latest application of the initialization pattern “11” (shown in green) is relevant and sufficient to charge the internal lines to the required values. Thus, even though the initial signal values in Figure 4.16b are “10” instead of the initialization pattern “11”, the fault can still be tested.

As was discussed in the previous section, the vector $stableWeak_i$ is utilized to model the point in time from which on the input i does not change anymore and remains stable. Hence, the second condition (no glitch-invalidations) is covered by these variables.

For the first condition (initialization) a vector $init$ with the same size as $Tvars$ is introduced. Unlike $stableWeak_i$, the vector $init$ is only encoded once for the entire fault-affected cell. When the variable $init[t]$ is assigned to $true$, the initialization condition is applied to the cell at time step t. This is modeled as follows:

$$\bigwedge_{t=t_{first}}^{t_{last}} \bigwedge_{i=0}^{\#inputs} init[t] \Rightarrow (\neg)Tvars_i[t] \tag{4.20}$$

The exact encoding of $init$ depends on the necessary initialization condition: When the input i needs to be ‘1’ for the initialization, the negation in front of the $Tvars_i$ is omitted, otherwise it is added. As an example, consider the clauses in Figure 4.17 that encode the initialization condition for the $NAND$-cell with the initialization pattern “11” from the previous example.

$$\big(\neg init[t_{first}] \lor Tvars_0[t_{first}]\big) \tag{4.21}$$
$$\big(\neg init[t_{first}] \lor Tvars_1[t_{first}]\big) \tag{4.22}$$
$$\big(\neg init[t_{first}+1] \lor Tvars_0[t_{first}+1]\big) \tag{4.23}$$
$$\big(\neg init[t_{first}+1] \lor Tvars_1[t_{first}+1]\big) \tag{4.24}$$
$$\dots$$
$$\big(\neg init[t_{last}-1] \lor Tvars_0[t_{last}-1]\big) \tag{4.25}$$
$$\big(\neg init[t_{last}-1] \lor Tvars_1[t_{last}-1]\big) \tag{4.26}$$
$$\big(\neg init[t_{last}] \lor Tvars_0[t_{last}]\big) \tag{4.27}$$
$$\big(\neg init[t_{last}] \lor Tvars_1[t_{last}]\big) \tag{4.28}$$

Figure 4.17.: List of the clauses encoding the $init$ vector for the initialization pattern “11”.

At this point the weak stability condition and the initialization are modeled with the vectors $stableWeak_i$ and $init$. These vectors are now combined to indicate whether the applied input values are “valid”. The input values are considered to be valid when they satisfy both of the initially defined conditions: Initialize the fault and ensure stability from the last initialization onwards.

To encode validity yet another variable vector, *valid*, with the size of $Tvars$ is utilized. The variable $valid[t]$ encodes that both conditions are either satisfied at the time step t or that they were satisfied at a previous point in time:

$$valid[t_{first}] \Rightarrow \Big(init[t] \wedge \bigwedge_{i \in Stable\ inputs} stableWeak_i[t]\Big) \tag{4.29}$$

$$\bigwedge_{t=t_{first}+1}^{t_{last}} valid[t] \Rightarrow \left(valid[t-1] \vee \Big(init[t] \wedge \bigwedge_{i \in Stable\ inputs} stableWeak_i[t]\Big)\right) \tag{4.30}$$

Note that the weak stability condition is only added for inputs that actually have a stability requirement (which is defined in the detection library). By forcing $valid[t_{last}]$ to be *true* through an assumption, any satisfying assignment that the SAT solver finds will correspond to a valid test pattern. This test pattern not only creates a waveform at the faulty cell input which will initialize the fault but also ensures that the test is not invalidated through a glitch. The $valid[t_{last}]$ assumption replaces the hard assumptions for the input values in T_1 that were encoded in the timing-unaware TSOF ATPG described in Section 4.1. The assumptions for the application of the propagation pattern in T_2 remain unaffected by this change.

Figure 4.18 shows the modeled implications for the considered $NAND$-cell with detection pattern "11" and a stability condition for the first input.

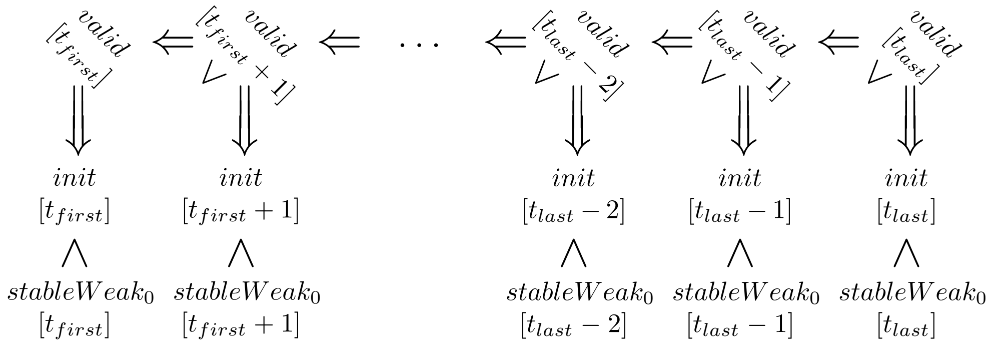
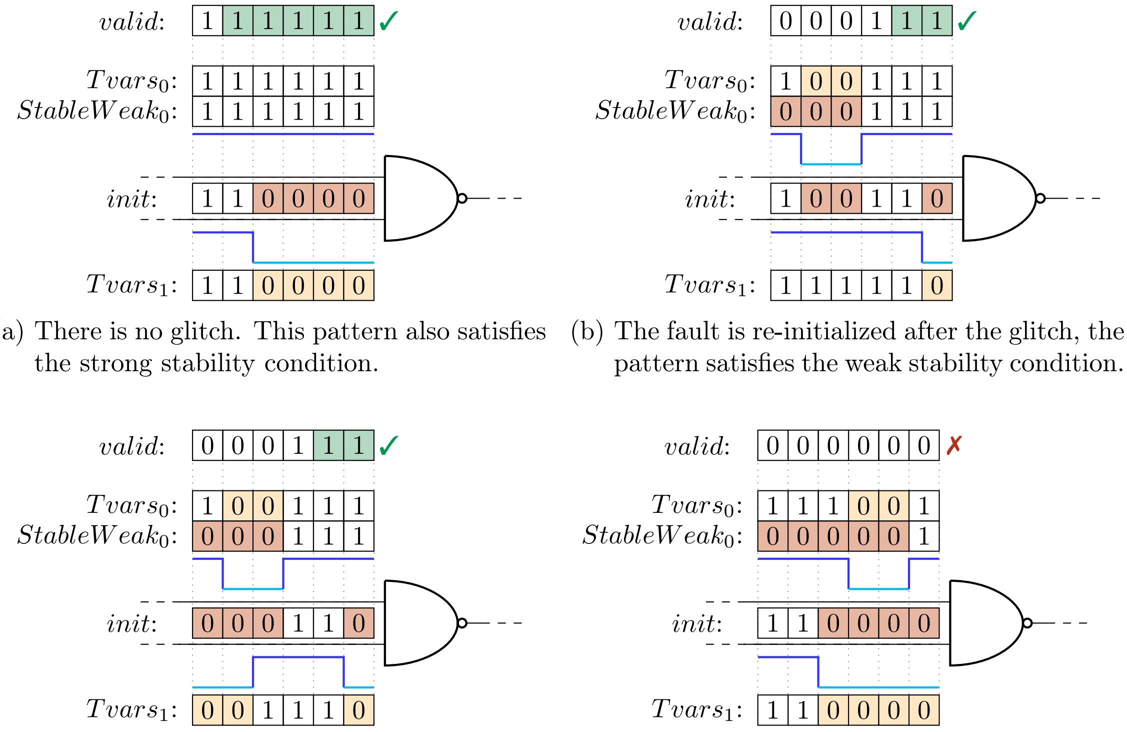

$valid[t_{first}]$
$valid[t_{first}+1]$
$\Leftarrow$
$\cdots$
$valid[t_{last}-2]$
$valid[t_{last}-1]$
$valid[t_{last}]$
$init[t_{first}] \wedge stableWeak_0[t_{first}]$
$init[t_{first}+1] \wedge stableWeak_0[t_{first}+1]$
$init[t_{last}-2] \wedge stableWeak_0[t_{last}-2]$
$init[t_{last}-1] \wedge stableWeak_0[t_{last}-1]$
$init[t_{last}] \wedge stableWeak_0[t_{last}]$

Figure 4.18.: The implications that model whether the applied input assignment is valid. $valid[t]$ can be assigned to *true* if one of the outgoing implications is satisfied.

Although the proposed modeling is rather complex and necessitates the addition of multiple new variable vectors, the overall cost is still very low. This is because all of the presented clauses are only required once and only for the fault-affected cell. For all other cells, the normal timing-aware modeling is sufficient. Furthermore, all of the presented formulas are designed to be easy to solve by utilizing only implications. If, for example, the signal is initialized and stable at time step $t_{last}-1$, the solver does not need to consider any additional stability constraints or initialization computations for the preceding time steps. The corresponding variables can simply be set to $false$.

While the presented encoding was developed with weak stability and glitch-initialization in mind, it of course also permits a normal initialization without any glitches and strong stability at the inputs. Figure 4.19 gives a final overview of the assignments of the $stableWeak_i$, $init$ and $valid$ variables for different input assignments. The stability variables are only encoded for the first input since the second input does not have a stability requirement. Fields marked in red are blocking $valid$ from being assigned to ‘1’ in this time step. Similarly, a yellow field blocks $init$ from being assigned to $true$. At time t, $init[t]$ can only be assigned to $true$ when there is no yellow field at t. Accordingly, $valid[t]$ can only be assigned to $true$ when there is no red field at t. A green marking in $valid$ indicates that $valid$ was assigned to $true$ in a previous time step and will, therefore, simply stay $true$.

Clearly, the proposed glitch modeling accurately reflects whether the fault effect will be masked. The test pattern that produces the invalid input assignment of Figure 4.19d would not be generated by the ATPG algorithm since the fault effect is masked by the glitch. This case is only shown for further clarification.



(a) There is no glitch. This pattern also satisfies the strong stability condition.

(b) The fault is re-initialized after the glitch, the pattern satisfies the weak stability condition.

(c) The fault is initialized because of the glitch at the second input.

(d) The glitch occurs after the initialization and hides the fault effect.

Figure 4.19.: Examples of how the proposed modeling of glitches handles the different ways a faulty cell might be initialized.

4.5. Charge-Sharing Mitigation

Resolving a charge-sharing conflict involves charging a certain line in the circuit to a special value before the propagation pattern is applied. In Section 4.3.1 two different mitigation techniques for charge-sharing conflicts were proposed: ensuring a certain switching order or utilizing a glitch. Both techniques aim at providing a special detection pattern for the faulty cell that will ensure that the line is correctly charged.

In this section, the encoding of the two mitigation techniques into a Boolean formula is discussed. Just like before, the goal is to teach the ATPG algorithm to only produce patterns that mitigate the charge-sharing conflict by one of the proposed methods. The necessary mitigation procedure is stored along with the detection pattern in the detection library just like the stability requirements that were encoded in the previous section. The discussed techniques are based on the encoding of the weak stability condition and assume that the corresponding variables are present in the Boolean formula.

4.5.1. Switching Order

In the switching order mitigation technique, the conflicting line is charged by ensuring that the inputs of the faulty cell switch in a certain order. This broad requirement is further refined by only considering detection patterns where switching a single input first is sufficient to resolve the charge-sharing conflict.

Assume that input i is the input that has to switch first. For a valid test there must be a time t where all inputs still have their T_1 value and at $t+1$ all inputs but i still have their T_1 value while i already has its T_2 value. Figure 4.20 shows an example for the mitigation of a charge-sharing conflict during the test of an OR-AND-Invert-cell with the detection pattern $\langle 1100, 1010 \rangle$ by switching the input $A2$ first. The green area marks the two time steps t and $t+1$.

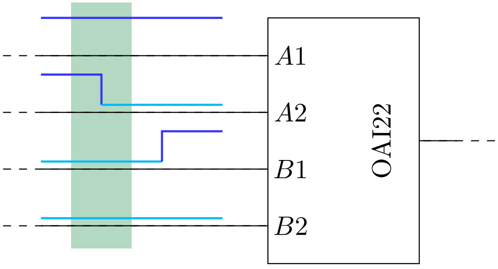


Figure 4.20.: Mitigating a charge-sharing conflict during the test of an AND-OR-Invert-cell by switching the input $A2$ first.

To incorporate the switching order condition into the Boolean formula of the solver, two new vectors with the same size as $Tvars$ are required.

The first new vector, *switchAct*, is built up just like the *init* vector in the previous section. However, instead of indicating whether the initialization pattern is applied to the circuit in time step t, *switchAct*$[t]$ indicates that at time step t all inputs apart from i still have their T_1 value whereas i has already switched to its T_2 value.

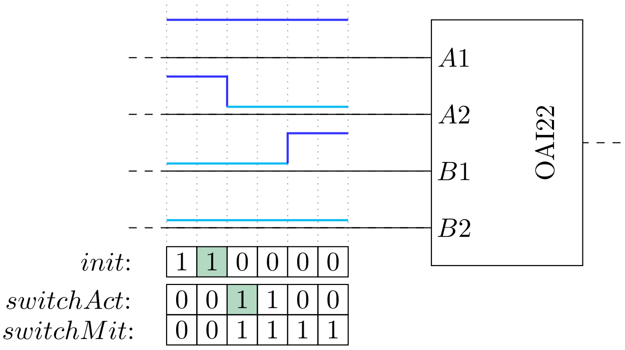
The second new variable vector, $switchMit$, encodes whether the charge-sharing conflict was successfully resolved. When $switchMit[t]$ is assigned to $true$, the conflict was either resolved in a previous time step or the correct switching order is applied at the current time step:

$$switchMit[t_{first}] = false \tag{4.31}$$

$$\bigwedge_{t=t_{first}+1}^{t_{last}} switchMit[t] \Rightarrow \Big(switchMit[t-1] \vee \big(init[t-1] \wedge switchAct[t]\big)\Big) \tag{4.32}$$

Figure 4.21 shows how the variable vectors are assigned for the example from Figure 4.20. The fields marked in green are those that are relevant for a successful charge-sharing mitigation.

By adding the assumption that $switchMit[t_{last}]$ must be $true$ when solving the formula, it is assured that any generated pattern will resolve the charge-sharing conflict through the correct switching order.



$init$:	1	1	0	0	0	0
$switchAct$:	0	0	1	1	0	0
$switchMit$:	0	0	1	1	1	1

Figure 4.21.: The assignment of the new variables when encoding a switching order initialization.

4.5.2. Glitch-Charging

Utilizing a glitch to charge the line that causes the charge-sharing conflict is similar to the glitch-initialization that was discussed in Section 4.4.2. However, there are two differences: Firstly, the glitch that resolves the charge-sharing conflict can occur at any time, independent from the stability of any other line. Secondly, the required signal values are different than the initialization values. The resolution of the charge-sharing conflict must occur before the fault is initialized for the last time.

These conditions are encoded by two new variable vectors with the same size as $Tvars$. The first vector, $glitchAct$ is similar to the vector $switchAct$ used for the switching order activation and is $true$ at time step t when the cell inputs have the values that are required to resolve the charge-sharing conflict. The second new variable vector, $glitchMit$, is assigned to $true$ when the charge-sharing conflict was either resolved at an earlier time step, or when the current input assignment at time step t corresponds to the required assignment that resolves the conflict:

$$glitchMit[t_{first}] \Rightarrow glitchAct[t_{first}] \tag{4.33}$$

$$\bigwedge_{t=t_{first}+1}^{t_{last}} glitchMit[t] \Rightarrow \Big(glitchMit[t-1] \vee glitchAct[t]\Big) \tag{4.34}$$

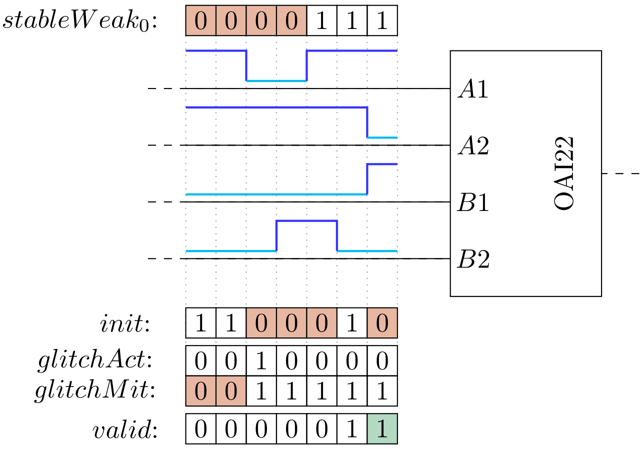
To ensure that the charge-sharing conflict is solved before the initialization pattern is applied for the last time, *glitchMit* is added as a requirement to *valid*:

$$\bigwedge_{t=t_{first}}^{t_{last}} valid[t] \Rightarrow glitchMit[t] \tag{4.35}$$

Therefore, to assign *valid* to *true* the charge-sharing conflict must have been previously resolved.

Figure 4.22 shows an example of the mitigation of a charge-sharing conflict during the test of a TSOF in an OR-AND-Invert-cell with the detection pattern $\langle 1100, 1010\rangle$ through a glitch at the input $A1$. For this final example it is further assumed that there is a stability condition for input $A1$. For this reason, all of the variables of the weak stability encoding are shown. Fields marked red are blocking $valid[t]$ from being assigned to *true* at the corresponding time step.

Note that in this example the initialization pattern is applied to the cell twice. However, the first time the stability requirement for input $A1$ is not satisfied. Furthermore, the glitch on input $B2$ – although not violating any stability requirements – is delaying the application of the initialization pattern by one time step. If this glitch would be any longer, the test would be unsuccessful because the initialization pattern could not be applied at all.



$stableWeak_0$:	0	0	0	0	1	1	1
init:	1	1	0	0	0	1	0
glitchAct:	0	0	1	0	0	0	0
glitchMit:	0	0	1	1	1	1	1
valid:	0	0	0	0	0	1	1

Figure 4.22.: Mitigating a charge-sharing conflict during the test of an AND-OR-Invert-cell by a glitch on input $A1$.

4.6. Minimum Event Durations

So far, all of the glitch-based techniques were considered to work instantaneously: A glitch immediately initializes the fault and a line that causes a charge-sharing conflict is charged immediately when the right pattern is applied to the inputs. As a result, detection patterns where very short glitches are responsible for the necessary effect are considered as valid patterns. Similarly, patterns in which the initialization pattern is only applied for a short time before the signals switch again are considered to be valid.

These effects can be observed in the previous example in Figure 4.22: The glitch on input $A1$ which mitigates the charge-sharing conflict only has a duration of two time steps. Furthermore, the initialization pattern is only applied for a single time step before the inputs switch to the propagation pattern. This might not be sufficient.

While this picture is accurate on an abstract circuit, it is more difficult in reality.

On the part of very short glitches, there are three main problems: Firstly, a very short glitch might simply not cause the desired effect to occur because not enough charge is transferred. Secondly, very short glitches might not be propagated through a real circuit due to the inherent delays of the cells and basic laws and limitations of physics. Thirdly, slight variations in the circuit's timing – which always occur due to process variations [74], [75] – might result in a timing in which the glitch does not even occur at all.

When the initialization pattern is applied for only a short amount of time, the desired effect might not occur because there might not have been enough time to transfer the needed charge. Furthermore, a slight variation in the timing might change the arrival times of the transitions and the initialization pattern might not be applied to the cell at all.

To tackle these challenges, the previous glitch-based requirements are extended with minimum duration requirements. These requirements ensure that a valid test pattern generates the desired effect for at least a minimum amount of time. Although this is no guarantee that process variations definitely will not invalidate the test pattern, the probability of this to occur is much lower.

4.6.1. Initialization

The minimum duration condition for the initialization is encoded directly into the *init* variable. This encoding covers both short glitch-initializations and short duration initialization pattern applications. With regard to the latter it should be noted that this problem only occurs in combination with the weak stability condition and *init* is, therefore, always present. For the strong stability condition it is ensured that the faulty cell is initialized in the first time frame and never discharged again by the switching activity. Hence, the initialization pattern is definitely applied for a sufficiently long time.

In the original encoding, when $init[t]$ was assigned to $true$, all of the input $Tvars$ had to have the initialization value. This definition is further extended by enforcing that the inputs must have had the initialization value for the past δ time steps as well. Thus, the encoding of *init* becomes:

$$\bigwedge_{t=t_{first}}^{t_{last}} \bigwedge_{i=0}^{\#inputs} init[t] \Rightarrow \bigwedge_{t_2=t-\delta}^{t} (\neg)Tvars_i[t_2] \tag{4.36}$$

4.6.2. Charge-Sharing Mitigation

Similar to the minimum duration condition for the initialization, the minimum duration condition for the charge-sharing mitigation techniques is encoded by modifying the definition of the previously defined variables. The goal is to ensure that the conflicting line is charged for at least μ time steps.

Switching Order

For the switching order mitigation technique, the variables *switchMit* are modified. Instead of only requiring that the necessary intermediate pattern (encoded by *switchAct*) is active at the current time step, it has to be active for the last μ time steps. In addition, *init* has to be *true* at the time step $\mu - 1$, which is only the case if the signals had the initialization value for the past δ time steps. Thus, *switchMit* is redefined as follows:

$$switchMit[t_{first}] = \cdots = switchMit[t_{first} + \mu] = false \tag{4.37}$$

$$\bigwedge_{t=t_{first}+\mu+1}^{t_{last}} switchMit[t] \Rightarrow \Big(switchMit[t-1] \vee \Big(init[t-\mu-1] \wedge \bigwedge_{t_2=t-\mu}^{t} switchAct[t_2]\Big)\Big) \tag{4.38}$$

Glitch Charging

For the glitch charging mitigation technique, it has to be ensured that the glitch that charges the conflicting line has a certain length. To this end, the definition of the variables *glitchAct* is modified. Recall that *glitchAct* works similar to *init* but instead of requiring the initialization pattern to become *true*, it enforces the input assignment that mitigates the charge-sharing conflict. For this condition to hold, the definition is modified similar to that of *init*. This ensures that the $Tvars$ have the correct value for the last μ time steps.

4.7. Glitch- and Charge-Sharing Aware TSOF ATPG

In this chapter, a multitude of different techniques to solve the challenges that arise when testing transistor stuck-open faults were introduced. All of these techniques are combined into an ATPG flow that attempts to create a test pattern for every possible TSOF with as little computational effort as possible. Section 4.7.1 introduces the basic structure of the proposed efficient timing-aware ATPG. The subsequent Section 4.7.2 discusses the overall ATPG flow used for the test pattern generation.

4.7.1. Timing-Aware ATPG

The timing-aware ATPG is based on the timing unaware ATPG that was presented at the beginning of this chapter. However, unlike this basic ATPG all of the timing information of the circuit needs to be encoded into the ATPG model to accurately reason about glitches and charge-sharing and to encode all of the advanced techniques presented in the previous sections. It would be possible to simply model the entire circuit with waveform accuracy (as presented in Section 2.2.7). This would result in a very complex circuit model and very large formulas.

On further analysis, it becomes apparent that timing information is not required for most parts of the circuit. Only in the justification cone of the fault the timing information is relevant. Only here the exact arrival time of a transition as well as glitches, can actually have an influence on the testability of a fault. Once the faulty cell was initialized and activated, the fault effect will definitely be propagated no matter what kind of glitches occur in the propagation or support cone.

Therefore, the ATPG utilizes an efficient hybrid circuit encoding: The justification cone of the cell is implemented with waveform accuracy, whereas all of the other parts of the circuit are modeled conventionally. The resulting formula is much smaller in size but still represents the exact same problem as it would have had the entire circuit been modeled with timing-awareness.

Transferring Timing Information to the Mapped Circuit

To handle complex cells in the circuits, in the proposed TSOF ATPG all cells are replaced by their logic gate definition. While this produces a mapped circuit with the same functionality, the timing information would be lost. To preserve the timing information it has to be transferred to the logic gates as well.

Figure 4.23 shows the transfer of the timing information of a multiplexer to its gate definition. The arrows indicate the delay from the corresponding input to the output of the cell or gate. If no arrow is shown, the delay is assumed to be 0. For each path from the input to the output of the cell, the timing is transferred to the first gate on that path in the gate definition. If multiple paths exist, the timing is added to all of them. After the transfer, the mapped circuit behaves just like the original circuit.

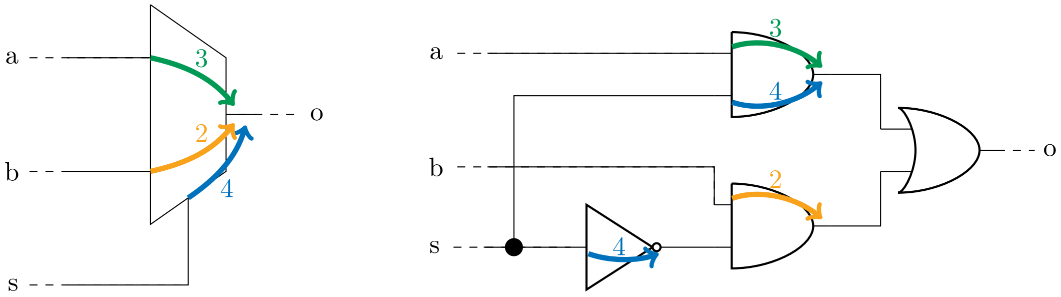


Figure 4.23.: Transferring the timing of a complex cell to its logic gate definition.

Modeling a Pre-Charging Time Frame

One of the proposed charge-sharing resolution techniques utilizes a third time frame to charge the conflicting line to the required value. This time frame is modeled by unrolling the circuit one more time. Since the pre-charging pattern in the time frame T_0 is created to be valid even in the presence of glitches, it is sufficient to model this unrolled circuit copy without timing information.

Figure 4.24 shows the two areas of the circuit that need to be modeled for the additional pre-charging time frame. In addition to the justification cone, which is needed to ensure that the fault is initialized, this is the support cone for the LOC type test.

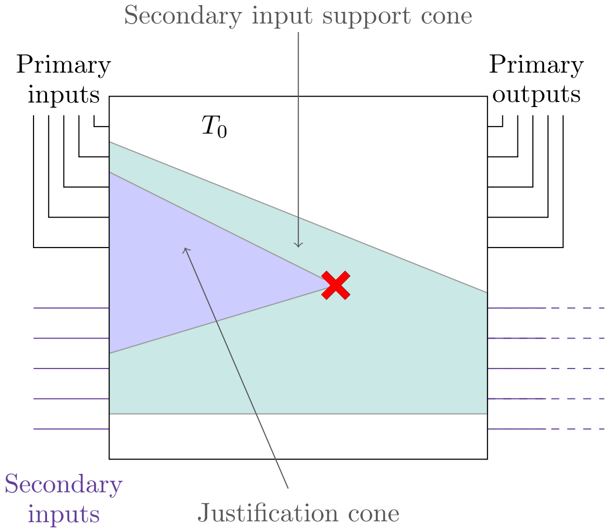


Figure 4.24.: Only the marked areas of the circuit are required when modeling an additional pre-charging time frame.

4.7.2. ATPG Flow

The overall aim of the presented glitch and charge-sharing aware transistor stuck-open fault ATPG is to compute a test pattern for a TSOF with as little effort as possible. Usually, there are multiple different detection patterns for a fault each of which has its own set of requirements of different difficulties and modeling cost. Since it is sufficient to find one test pattern for a fault, the detection patterns are tried in ascending order of difficulty. Figure 4.25 shows the overall ATPG flow.

Detection patterns without any charge-sharing conflicts can be modeled more easily and are less complex to compute. These are considered first. If the detection pattern does not have any stability requirements, it is handled by the timing-unaware ATPG algorithm presented in Section 4.1 which is marked as node 1 in Figure 4.25. Otherwise, the timing-aware ATPG first attempts to find a test pattern which satisfies the strong stability condition (node 2). If there is no test pattern that satisfies the strong stability condition, weak stability (node 3) and finally glitch-initialization are considered (node 4).

Secondly, detection patterns with a charge-sharing conflict that can be mitigated are

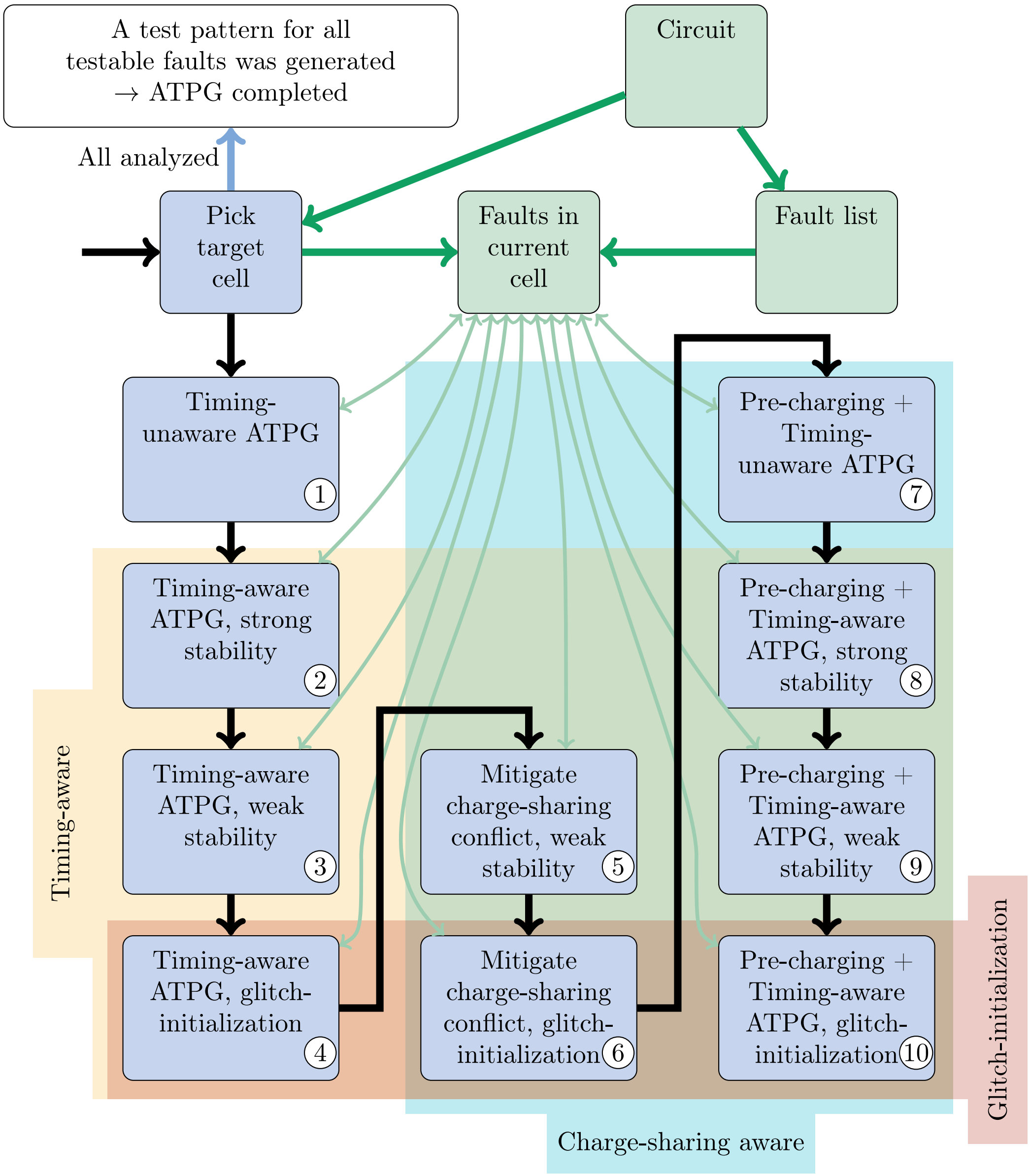

A test pattern for all testable faults was generated → ATPG completed
Circuit
All analyzed
Pick target cell
Faults in current cell
Fault list
Timing-unaware ATPG 1
Pre-charging + Timing-unaware ATPG 7
Timing-aware ATPG, strong stability 2
Pre-charging + Timing-aware ATPG, strong stability 8
Timing-aware
Timing-aware ATPG, weak stability 3
Mitigate charge-sharing conflict, weak stability 5
Pre-charging + Timing-aware ATPG, weak stability 9
Timing-aware ATPG, glitch-initialization 4
Mitigate charge-sharing conflict, glitch-initialization 6
Pre-charging + Timing-aware ATPG, glitch-initialization 10
Glitch-initialization
Charge-sharing aware

Figure 4.25.: Overview of the complete TSOF ATPG flow. The list of undetected faults in the current cell is continuously updated to avoid the creation of multiple test patterns for the same fault. The black arrows indicate the program flow, the green arrows the flow of information.

considered. Again, the ATPG first attempts to create patterns with the weak stability condition (node 5) and only considers glitch-initialization (node 6) when the previous approach fails. Note that the glitch-mitigation techniques are based on the modeling of the weak stability condition. Hence, the strong stability condition is not applicable in combination with these techniques and is skipped. However, any pattern that satisfies the strong stability condition also satisfies the weak stability condition. Thus, by only considering the weak stability condition, all possible test patterns are found nonetheless.

Finally, detection patterns with a charge-sharing conflict that can be resolved through a pre-charging time frame are considered. The ATPG flow is similar to that without any charge-sharing conflicts but with an additional initial time frame (nodes 7 to 10).

In addition to attempting to find a test pattern for the easiest detection pattern first, the ATPG also utilizes incremental solving. To this end, all of the faults in a cell are considered. In each previously discussed ATPG step, the generated Boolean formula is reused for every detection pattern with the same requirements that could detect an, as of yet, undetected fault in the cell. If a fault was detected by a previous step it is not considered any further. Thus, for every cell the ATPG creates at most ten different Boolean formulas, no matter how many different transistors (and, therefore, potential faults) the cell has.

The presented approach offers yet another benefit: By first considering detection patterns without stability requirements and strong stability, the ATPG creates test patterns which are more robust to process variations. Only afterwards the more complex, and potentially more vulnerable, glitch-based methods are used. The overall aim of the presented approach is to test as many TSOFs as possible. To this end, in addition to using modeling techniques of different granularity, it even utilizes glitch-initialization or a pre-charging time frame to test faults which would be considered to be untestable by a traditional ATPG approach.

4.8. Evaluation

This chapter introduced an advanced glitch and charge-sharing aware TSOF ATPG algorithm that contains ten different modes to handle the different requirements of a detection pattern while still creating a test pattern with as little computational effort as possible. In this section, a thorough evaluation of the algorithm is performed.

Section 4.8.1 highlights the general performance of the ATPG. In the subsequent sections, the different aspects of the ATPG are analyzed in detail. Section 4.8.2 focuses on the timing-awareness, Section 4.8.3 on glitch-initialization and Section 4.8.4 on charge-sharing. Furthermore, in Section 4.8.5 minimum event durations are analyzed. In Section 4.8.6 the importance of considering glitches and charge-sharing during the pattern generation is illustrated. Finally, Section 4.8.7 shows the influence of the utilized D-chain implementation on the solve speed.

All of the tables referenced in the evaluation are printed at the end of the chapter.

4.8.1. General ATPG Performance

Table 4.1 shows the overall results of the test pattern generation with a timeout of 10 s for each mode. Furthermore, Table 4.2 gives a more detailed look at the achieved fault coverage for the different modes of the TSOF ATPG which is summarized in Figure 4.26. To increase the readability, some of the modes are combined and more detailed results are given in the subsequent sections. The timing-aware modes with strong and weak stability (nodes 2 and 3 in Figure 4.25) are combined into the "glitch-free" category. Furthermore, all of the pre-charging techniques (nodes 7 to 10 in Figure 4.25) are combined into the "CS pre-charged" category.

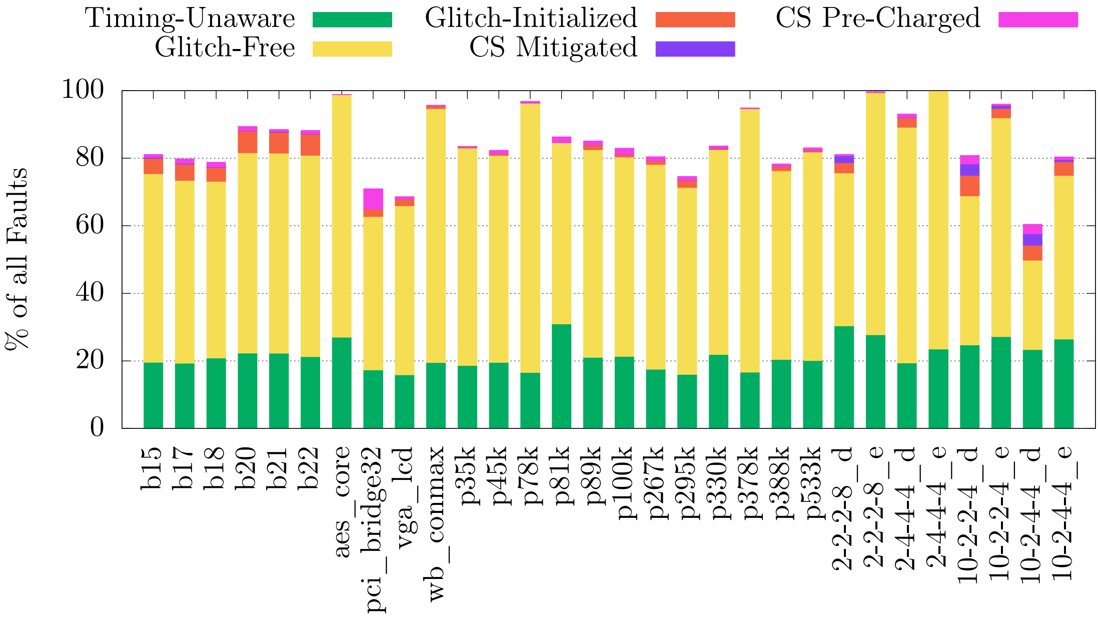


Figure 4.26.: Achieved **fault coverage** with the different TSOF ATPG modes.

On average across all circuits, about 21.6 % of faults can be tested by the timing-unaware ATPG because of a detection pattern without stability requirements. In addition, 59.4 % of faults can be tested by ensuring glitch freedom by a carefully crafted pattern through the timing-aware ATPG. The faults that are detected by the timing-unaware or normal timing-aware ATPG are considered to be conventionally testable as they do not require any advanced reasoning about glitches. This is because the initialization pattern and the propagation pattern are applied to the faulty cell in the normal manner.

Nonetheless, it is unlikely that all of these faults would actually be successfully tested by patterns generated by a timing-unaware ATPG algorithm because of glitch-invalidations. This topic is discussed further in Section 4.8.6.

The remaining faults are considered to be conventionally untestable faults as they cannot be tested by conventional test patterns. They might, however, be testable by actively utilizing glitches. The presented ATPG algorithm supports multiple methods to generate patterns for these faults which are beyond the reach of a conventional ATPG algorithm: By using advanced methods which utilize glitches or the switching order of signals to initialize the fault or to mitigate a charge-sharing conflict, another 2.3 % and 0.5 % (respectively) of faults can be tested. Finally, by extending the test pattern into a third time frame to pre-charge lines that cause a charge-sharing conflict, the fault coverage is increased by another 1.1 %.

In Figures 4.27 and 4.28 the average runtime per fault for the different modes is shown. This time includes both the formula generation time as well as the solve time itself. For the ITC'99, IWLS and NXP circuits the average runtime per fault is below 200 ms. For the AES circuits, on the other hand, the average runtime can be as high as almost 7 seconds per fault. The differences of the AES circuits will be discussed further at the end of this section.

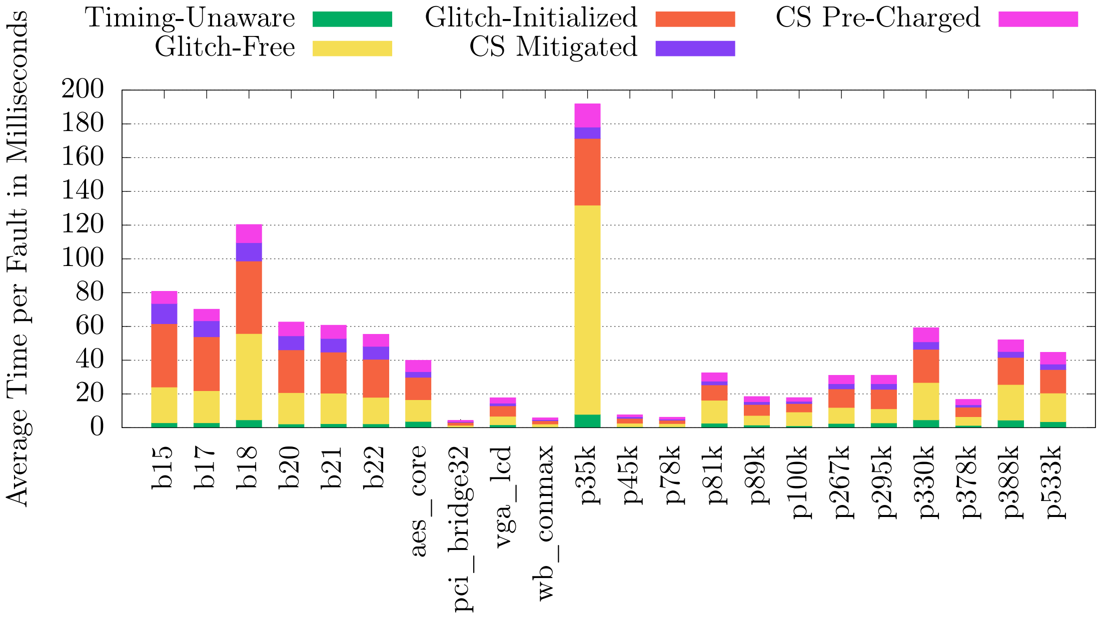


Figure 4.27.: **Average runtime** per fault for the ITC'99, IWLS and NXP circuits with the different TSOF ATPG modes.

Because of the much simpler formulas the average runtime of the timing-unaware ATPG mode is much shorter than that of the timing-aware modes. Nonetheless, the presented efficient hybrid circuit modeling still results in very fast computation times per fault given that the ATPG considers glitches and advanced initialization properties. Overall, the ATPG can quickly and successfully generate a test pattern even for the large industrial circuits (NXP).

Across all circuits, the number of timeouts (shown in the last column in Table 4.1) is very low. Thus, a test pattern can be computed for almost every fault. The algorithm did not find a test pattern due to a timeout for at most 0.03 % of faults.

The number of timeouts for the AES circuits is very low (two timeouts in total). Hence, although they are difficult to solve, the AES circuits can still easily be handled by the proposed algorithm.

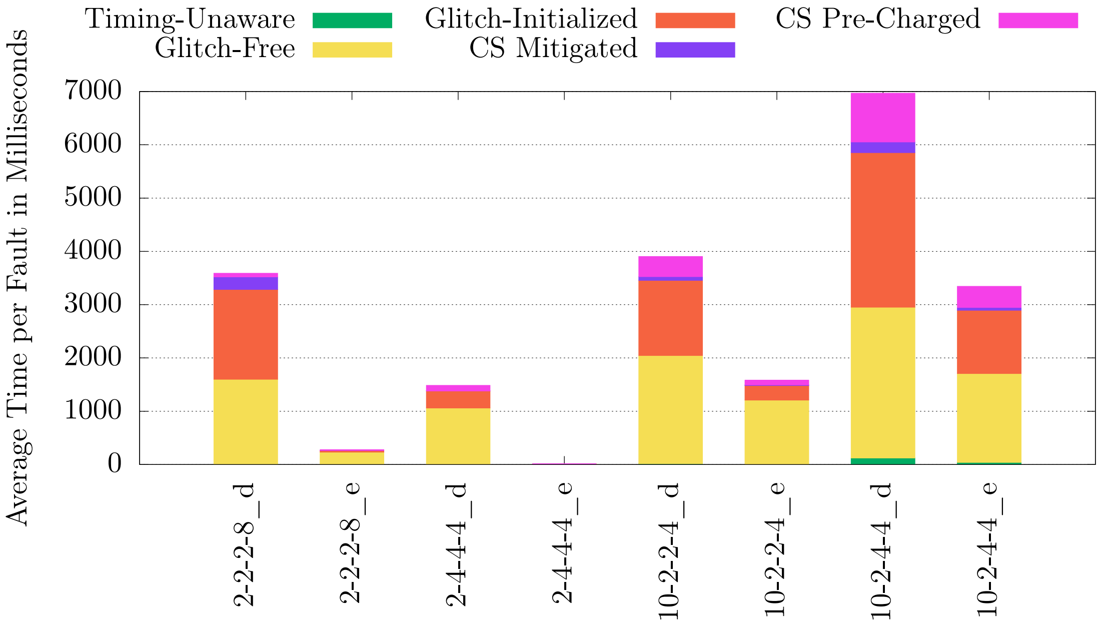


Figure 4.28.: **Average runtime** per fault for the AES circuits with the different TSOF ATPG modes.

AES Circuits

The AES circuits were generated for [W2] as artificial benchmarks for research purposes. The circuits are purely combinational and encode up to 10 rounds of a slightly reduced version of the advanced encryption standard. Therefore, the combinational complexity and path length is much higher than in the remaining benchmark and industrial circuits and the justification, support and propagation cone will cover a much larger portion of the circuit for most faults. Because of the high combinational complexity and many parallel paths which do not diverge, the window of possible activity (the distance between t_{first} and t_{last}) at the faulty cell is much larger. Thus, encoding the stability requirements becomes much more expensive and the formulas are harder to solve. Figure 4.29 shows the average and maximum number of $Tvars$ for each input with a stability requirement across all formulas and faults.

Clearly, the average number of $Tvars$ on these inputs is much higher for the AES circuits, often even surpassing the maximum number of $Tvar$s of other circuits. The most extreme case, 10-2-4-4_d, has about 507 $Tvars$ that represent the signals with a stability requirement on average. This is more than the maximum number of $Tvar$s for any other circuit class.

Because of the long paths without any flip-flops, the AES circuits might not represent high speed industrial designs. Nonetheless, such designs can occur when the maximum path length is not the limiting factor. The results show that even in such highly complex circuits the presented ATPG algorithm still successfully computes test patterns for the TSOFs.

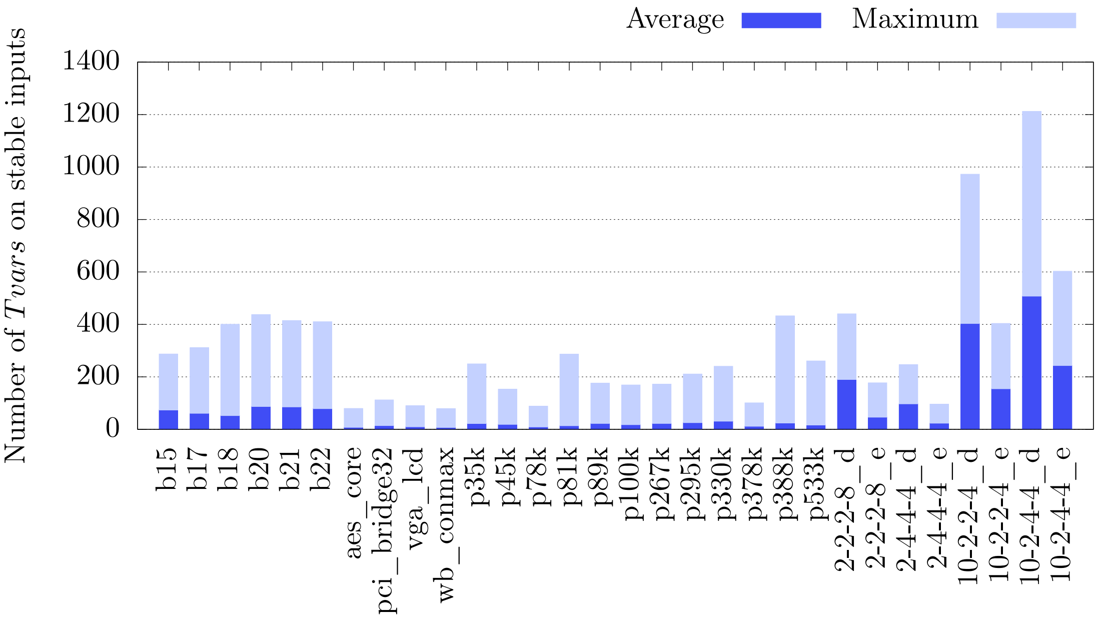


Figure 4.29.: Average and maximum **number of *Tvars*** for inputs with a stability requirement.

4.8.2. Timing-Aware ATPG

This section focuses on the coverage of faults with a stability requirement but without any charge-sharing conflicts. The test pattern generation is performed by the steps marked as nodes 2 and 3 in Figure 4.25. The fault coverage with the strong and weak stability condition is shown in Figure 4.30.

For most circuits, the vast majority of faults with a stability requirement can be tested by ensuring strong stability. Hence, there is a test pattern which does not generate any glitches at the stable inputs at all. For the AES circuits, on the other hand, this does not hold true: Here, many faults (on average 21.6 %) can only be tested when the more exact weak stability condition is utilized. For these faults there is no test pattern that ensures total glitch freedom at the stable inputs. However, there is a test pattern which still applies the initialization pattern after any dangerous glitches have subsided. For the remaining circuits, only 0.2 % of faults require the more exact modeling. This is, again, due to the more complex structure of the AES circuits which facilitates glitches.

4.8.3. Glitch-Initialization

By utilizing glitches to initialize faults the fault coverage can be extended beyond that of a traditional ATPG algorithm. Figure 4.31 shows the coverage of conventionally untestable faults through glitch-initialization.

On average, about 15,6 % of conventionally untestable faults can be tested by utilizing glitch-initialization. Generally, the coverage of the ITC'99 and AES circuits is higher than that of the remaining circuits. This is due to the larger window of activity at the faulty

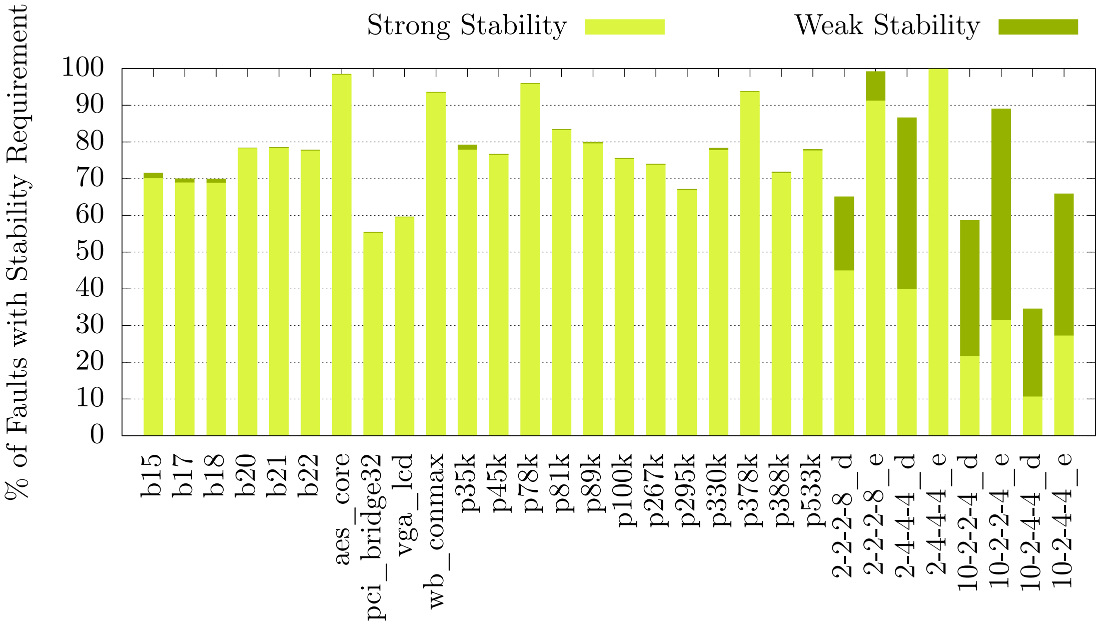


Figure 4.30.: **Coverage** of faults with a stability requirement.

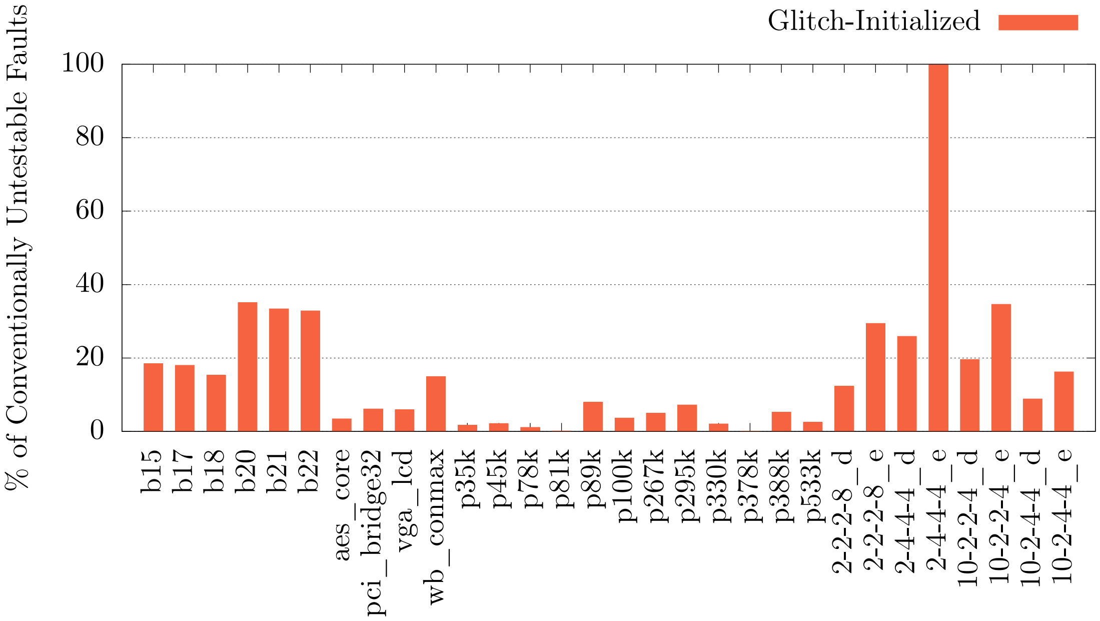


Figure 4.31.: **Coverage** of conventionally untestable faults through glitch-initialization.

cell: As shown in Figure 4.29, for these circuits the average number of $Tvar$s at the faulty cell's inputs is larger and, hence, more glitches can occur. Therefore, when utilizing the presented advanced approach to testing TSOFs, glitches can actually be beneficial and might turn otherwise untestable faults into testable ones.

Since it is possible to initialize the newly testable faults through a glitch, this initialization could also occur during the normal circuit application. Thus, simply considering them as untestable – like a conventional ATPG algorithm does – would increase the DPPM if the fault is not found by other means.

4.8.4. Charge-Sharing

In addition to avoiding dangerous glitches and utilizing beneficial ones for fault initialization, the presented TSOF ATPG algorithm also contains advanced means of mitigating charge-sharing conflicts and considers a pre-charging time frame to resolve the conflict. Figure 4.32 shows the coverage of conventionally untestable faults through the resolution techniques.

Depending on the circuit, up to 10 % of conventionally untestable faults can be tested through one of the two mitigation techniques (shown in dark violet). As Figure 4.33 shows, most of these faults become testable because the test pattern ensures a certain switching order, while the remaining 14.7 % utilize a glitch.

Furthermore, Figure 4.32 shows that up to 16 % of conventionally untestable faults can be tested by adding a pre-charging time frame (shown in pink).

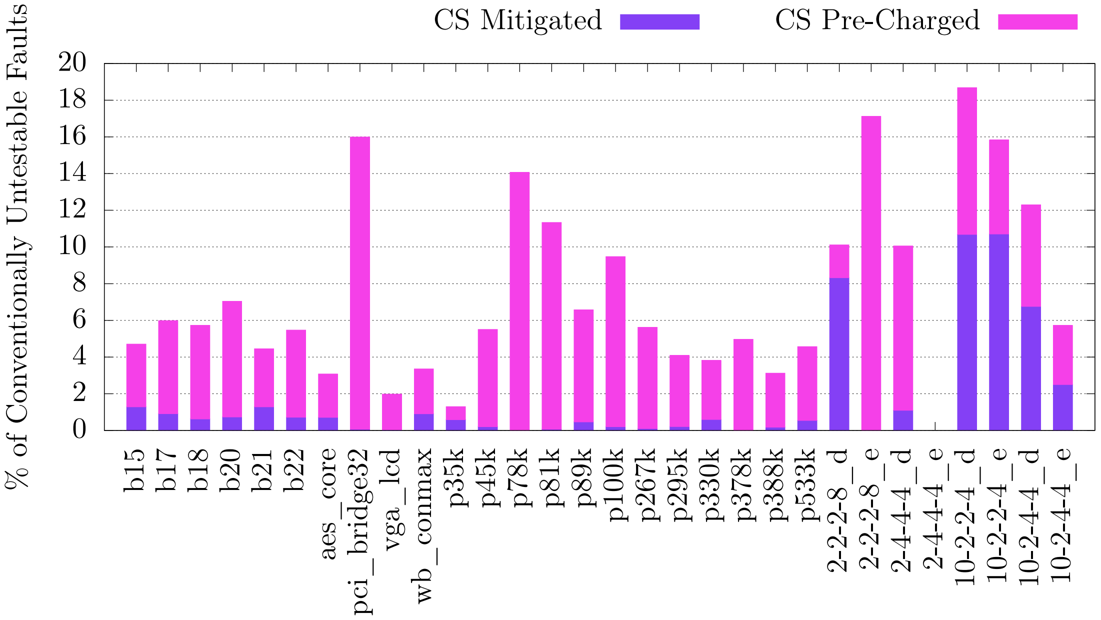


Figure 4.32.: **Coverage** of conventionally untestable faults by resolving the charge-sharing conflict.

Unlike dangerous glitches which have a high probability of invalidating a test by hiding the fault effect, a charge-sharing conflict might actually be unproblematic. This is espe-

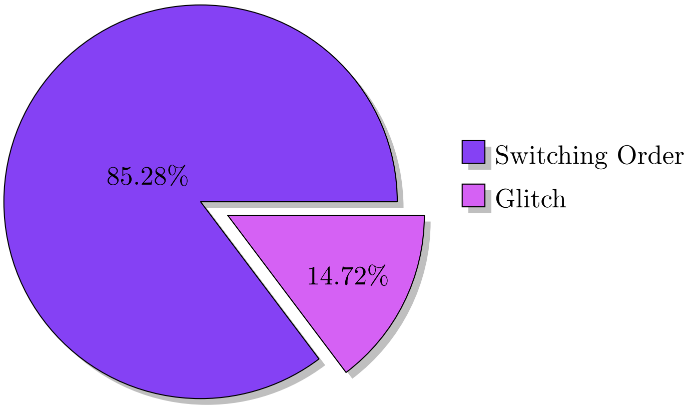


Figure 4.33.: Average **portion of charge-sharing sensitive faults** that can be tested because of the switching order or by glitches.

cially the case if the floating charge is stored by the output line. Since the output usually has a higher capacitance than any of the internal lines, sharing the charge between an internal line and the output will most likely have only a small effect on the floating charge.

While it is definitely beneficial to generate test patterns with charge-sharing awareness that are guaranteed to avoid any such conflict, one should also attempt to test the faults for which the conflict cannot be avoided. Figure 4.34 shows the overall fault coverage that is obtained by first utilizing the presented ATPG algorithm and then creating test patterns for all the previously untestable faults without considering charge-sharing. Overall, the fault coverage is increased by only about 0.7 %. on average. Therefore, for the vast majority of faults the charge-sharing conflict can be avoided by first utilizing the presented charge-sharing aware ATPG which guarantees a high quality of the generated patterns.

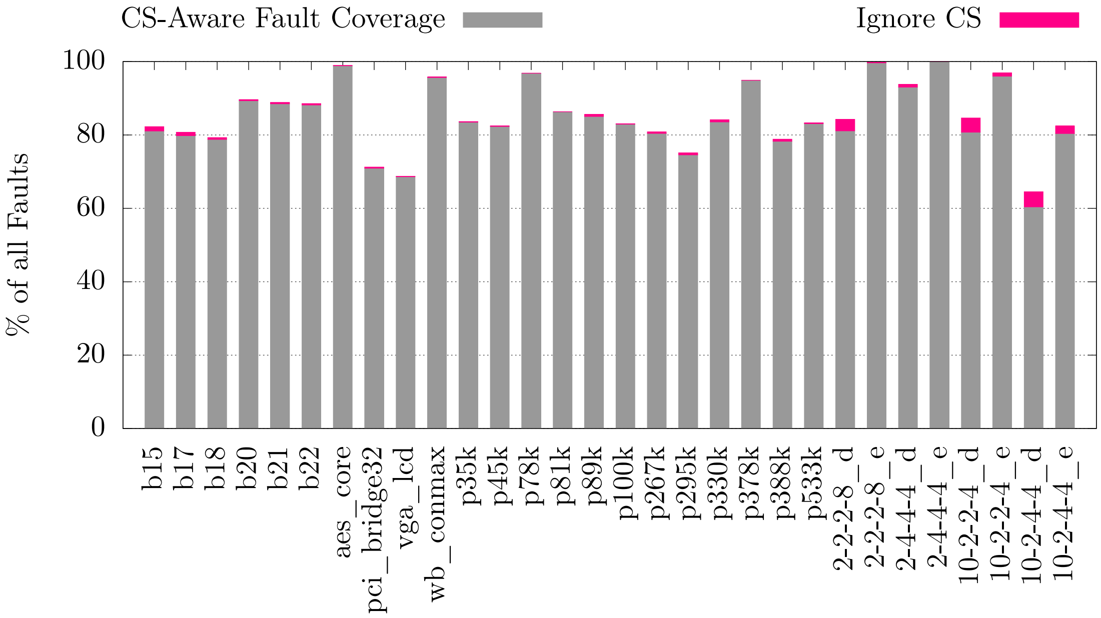


Figure 4.34.: Additional **fault coverage** when ignoring all charge-sharing conflicts.

4.8.5. Minimum Event Durations

So far, all of the generated test patterns were created without any minimum event duration requirements. The presented ATPG algorithm contains the parameters δ and μ to define the minimum fault initialization and charge-sharing mitigation length. By increasing the minimum event duration requirements, the generated test patterns will become more robust against slight changes in timing due to process variations. Figure 4.35 shows the effect of different minimum event durations on the overall fault coverage. For these experiments δ and μ are set to the same value.

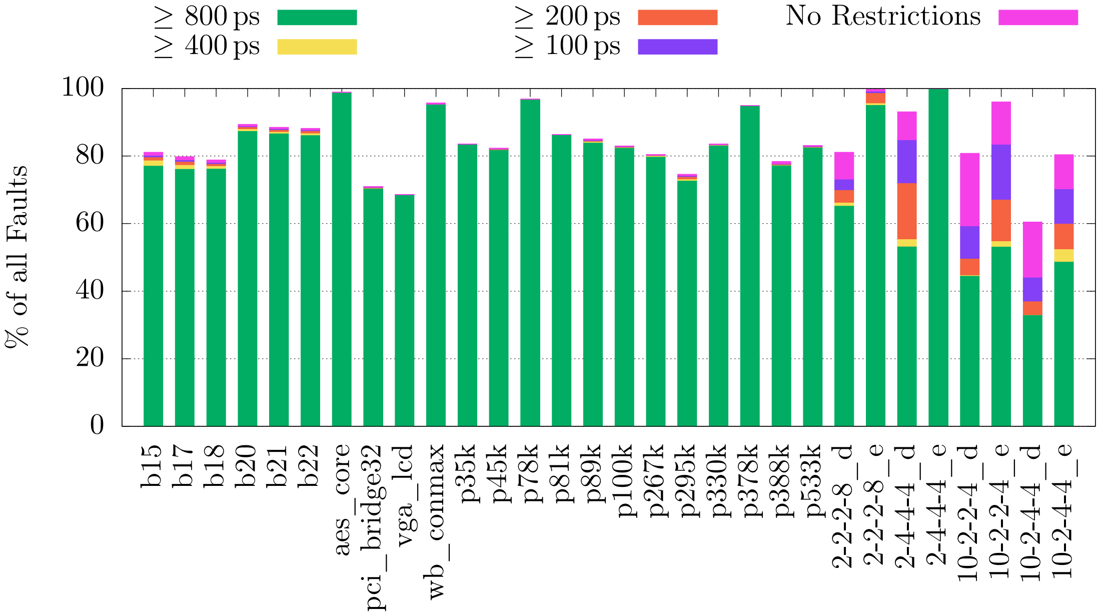


Figure 4.35.: Achieved **fault coverage** when enforcing different minimum event duration requirements.

As expected, a more robust pattern could not be generated for all of the detectable faults. Nonetheless, even for very long requirements, many faults still remain testable. In practice, a minimum duration requirement that is longer than the delay of the cell in question is probably not necessary. In the considered circuits, most cells have a delay of below 400 ps. With a minimum duration requirement of 400 ps the fault coverage is reduced by only 6.8 %. on average. The AES circuits are affected more strongly than the remaining circuits due to their higher combinational complexity which results in more glitches.

Similar to the charge-sharing requirements, it might be beneficial to first generate test patterns with a fitting minimum duration length requirement. In a second step, test patterns without any duration requirements for the remaining faults can be generated. While these test patterns are less robust, they might still successfully detect the fault. Furthermore, these patterns still outperform conventional patterns that were generated without glitch-awareness.

4.8.6. The Importance Of Considering Glitches and Charge-Sharing

One of the main innovations of the presented ATPG algorithm lies with its glitch-awareness. To evaluate the importance of considering glitches and charge-sharing during the test pattern generation the following experiment is performed: For each fault with a stability requirement ten different test patterns are generated through the timing-unaware ATPG without considering charge-sharing. These patterns might, therefore, result in malicious glitches or fail to charge all of the cell internal lines to the necessary value. In the next step, the generated patterns are tested for the occurrence of glitch-invalidations or charge-sharing conflicts.

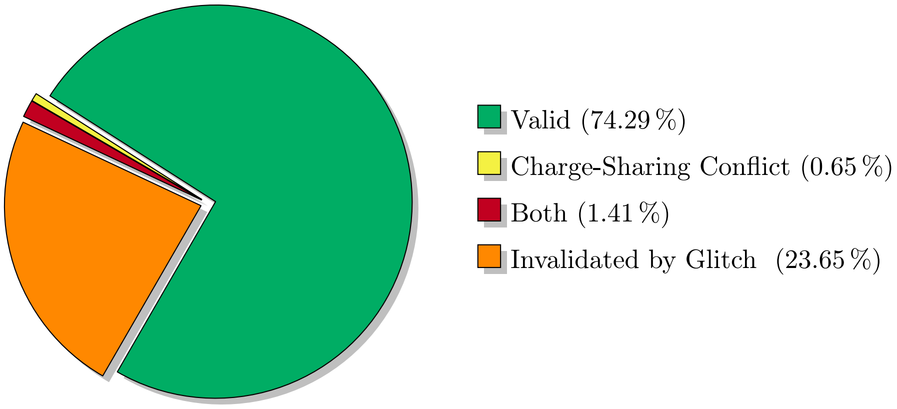


Figure 4.36.: **Share of test patterns**, generated without timing-awareness, that have a charge-sharing conflict or are invalidated by a glitch.

Figure 4.36 shows the share of patterns that were valid as well as the share of patterns that were invalidated by a charge-sharing conflict, by a glitch, or by both. For only about 74.3 % of the generated patterns it is ensured that the fault is definitely found. Of the remaining patterns, the 25.1 % that are potentially invalidated by a glitch are especially hazardous. While a charge-sharing conflict might not be a problem in a real circuit, a glitch-invalidation will most likely mask the fault effect. Thus, the real fault coverage is significantly reduced and the number of defective shipped parts significantly increased.

4.8.7. D-Chains in the TSOF ATPG

The previous chapter introduced and evaluated different D-chain variants in the scope of a SAT-based stuck-at ATPG. The lower complexity of the fault model allowed for a thorough evaluation of a large amount of different D-chains across all of the circuits. Since the presented TSOF ATPG is also SAT-based, the hybrid D-chain with the dynamic node selection heuristic was used to speedup the solve time. This D-chain was selected because it provided the fastest overall solve speed improvement in the stuck-at ATPG.

This section furthermore analyzes the solve time of the TSOF ATPG without any D-chains and with the backward D-chain. Due to the higher complexity and overall run times, the experiments are restricted to these two D-chains which had the highest gains on the stuck-at ATPG. Figure 4.37 shows the change in solve time for the two different D-chains in comparison to utilizing no D-chain at all.

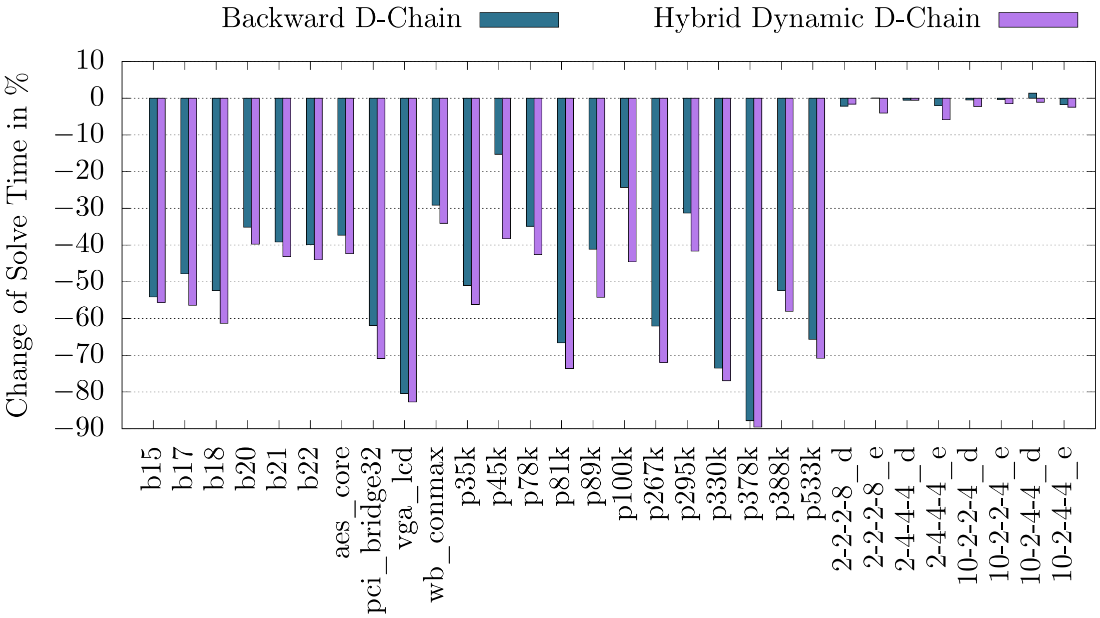


Figure 4.37.: **Change in solve time** when using the backward or the hybrid dynamic D-chain in comparison to using no D-chain.

The speedup of the D-chains is generally similar to that obtained in the stuck-at ATPG. Again, the hybrid D-chain with the dynamic node selection heuristic slightly outperforms the backward D-chain. On average across all circuits, the hybrid D-chain reduces the solve time by about 42 %.

For the AES circuits, the gains of the different D-chains are minimal. This can be explained with the much higher combinational complexity, again. The difficulty of the TSOF ATPG for these circuits lies not with the propagation of the fault but instead with the stability requirements and successful fault initialization.

Nonetheless, the overall results clearly show that considering D-chains is of great importance when creating an efficient SAT-based ATPG algorithm. The additional information appears to be vital no matter what fault model is used. This is not surprising as the propagation of the fault effect is the same no matter how the fault itself is defined.

4.9. Summary

This chapter presented and evaluated a SAT-based, glitch and charge-sharing aware transistor stuck-open ATPG algorithm. Conventionally, tests for TSOFs have been difficult to generate because of the possibility of glitch-invalidations and charge-sharing conflicts. The presented algorithm not only allows for the generation of glitch and charge-sharing aware test patterns but also utilizes glitches and the circuit's timing to increase the fault coverage beyond that possible with a traditional ATPG.

The key points of the chapter are the following:

- When testing TSOFs glitch-invalidations and charge-sharing conflicts have to be taken into account to ensure a successful test. These problems do not occur in the more basic stuck-at and transition-delay fault models (because they are defined at the circuit level) but are relevant for advanced fault models which take cell-internal effects into account.
- Considering glitches during the test pattern generation is of high importance as many (in the conducted experiments over 25 % on average) patterns that are generated without glitch-awareness are invalidated.
- The presented approach reliably and deterministically generates test patterns for the TSOFs. By utilizing a SAT-based hybrid modeling, incremental solving as well as by ordering the different ATPG modes in ascending order of difficulty, the approach is scalable even to large industrial circuits and highly complex combinational circuits.
- By accurately considering the arrival time of signals as well as glitches, the fault coverage can be extended beyond that of traditional ATPG algorithms.
- By requiring minimum event durations for glitches and the length of the fault initialization, the robustness against process variations can be increased. Such a more robust test pattern can be generated for almost all of the detectable faults.

This chapter not only presented a novel and highly effective ATPG algorithm but also highlighted the versatility and power of SAT-based modeling in general. The created model can easily be extended to incorporate additional requirements or can even be transformed to completely different fault models. This is especially important as many other advanced fault models (e.g., cell-aware test) might also be susceptible to glitch-invalidations or similar effects. In addition to analyzing different advanced fault models, the presented algorithm itself could be further improved with the addition of a fault simulator. This simulator would have to be timing-aware by itself to accurately model glitches and to consider charge-sharing conflicts.

Overall, the presented results show the importance of considering complex effects like glitch-invalidations and charge-sharing conflicts when utilizing advanced fault models. At the same time, they also highlight how the circuit's timing characteristics can be used to further increase the overall fault coverage through glitch-initializations and similar effects.

Table 4.1.: Results of the test pattern generation by the TSOF ATPG with a timeout of 10 s per fault.

	Circuit	# Faults	Runtime (s)	Memory (MB)	FC (%)	# Not Detected due to Timeout
ITC'99	b15	23 374	1 894.04	97.9	81.12	0 (0.00 %)
	b17	77 984	5 502.68	183.2	79.82	0 (0.00 %)
	b18	228 636	27 651.19	409.3	78.81	29 (0.01 %)
	b20	37 970	2 383.86	115.9	89.36	0 (0.00 %)
	b21	38 182	2 323.95	114.7	88.51	0 (0.00 %)
	b22	53 612	2 979.41	128.7	88.19	0 (0.00 %)
IWLS	aes_core	71 060	2 838.20	144.4	98.89	0 (0.00 %)
	pci_bridge32	58 514	267.06	168.5	70.97	0 (0.00 %)
	vga_lcd	315 264	6 232.50	815.7	68.62	0 (0.00 %)
	wb_conmax	108 444	629.55	228.2	95.68	0 (0.00 %)
NXP	p35k	74 230	14 258.72	212.7	83.54	4 (0.01 %)
	p45k	78 838	609.09	186.1	82.32	0 (0.00 %)
	p78k	209 950	1 311.30	330.2	96.85	0 (0.00 %)
	p81k	258 576	8 562.81	458.5	86.34	0 (0.00 %)
	p89k	170 922	3 177.08	365.1	85.08	0 (0.00 %)
	p100k	182 354	3 318.53	440.4	82.95	2 (0.00 %)
	p267k	377 210	12 045.10	769.7	80.45	2 (0.00 %)
	p295k	432 420	13 926.36	912.5	74.60	0 (0.00 %)
	p330k	395 182	23 843.88	799.3	83.57	114 (0.03 %)
	p378k	1 088 518	18 674.85	1 503.7	94.92	0 (0.00 %)
	p388k	843 628	45 153.60	1 527.4	78.30	10 (0.00 %)
	p533k	1 339 698	63 130.71	2 313.2	83.10	6 (0.00 %)
AES	2-2-2-8_d	33 996	122 026.46	165.0	81.13	2 (0.01 %)
	2-2-2-8_e	30 588	8 448.61	92.0	99.67	0 (0.00 %)
	2-4-4-4_d	15 174	22 552.60	55.8	93.07	0 (0.00 %)
	2-4-4-4_e	12 390	160.70	50.6	100.00	0 (0.00 %)
	10-2-2-4_d	12 768	49 818.25	163.4	80.78	0 (0.00 %)
	10-2-2-4_e	13 144	20 785.92	51.9	96.02	0 (0.00 %)
	10-2-4-4_d	23 228	161 871.10	190.3	60.43	0 (0.00 %)
	10-2-4-4_e	24 236	81 064.15	88.4	80.41	0 (0.00 %)

Table 4.2.: Fault coverage for the different modes of the TSOF ATPG.

	Circuit	Fault Coverage (%)				
		Timing-Unaware	Glitch-Free	Glitch-Initialized	CS Mitigated	CS Pre-Charged
ITC'99	b15	19.61	55.81	4.53	0.32	0.84
	b17	19.34	54.10	4.78	0.24	1.35
	b18	20.86	52.31	4.11	0.17	1.37
	b20	22.34	59.29	6.44	0.13	1.16
	b21	22.29	59.22	6.17	0.24	0.58
	b22	21.26	59.60	6.28	0.14	0.91
	Average	20.95	56.72	5.39	0.21	1.03
IWLS	aes_core	27.02	71.79	0.04	0.01	0.03
	pci_bridge32	17.36	45.38	2.27	0.02	5.93
	vga_lcd	15.90	50.04	2.00	0.02	0.66
	wb_conmax	19.51	75.21	0.79	0.05	0.13
	Average	19.95	60.60	1.28	0.02	1.69
NXP	p35k	18.65	64.37	0.29	0.10	0.12
	p45k	19.57	61.28	0.41	0.04	1.01
	p78k	16.60	79.68	0.04	0.00	0.52
	p81k	30.98	53.59	0.01	0.01	1.74
	p89k	21.05	61.49	1.38	0.08	1.07
	p100k	21.35	59.04	0.71	0.04	1.82
	p267k	17.52	60.62	1.08	0.02	1.21
	p295k	16.02	55.35	2.06	0.06	1.11
	p330k	21.89	60.66	0.35	0.10	0.56
	p378k	16.67	77.99	0.00	0.00	0.27
	p388k	20.42	55.91	1.23	0.04	0.70
	p533k	20.12	61.69	0.45	0.10	0.73
	Average	20.07	62.64	0.67	0.05	0.90
AES	2-2-2-8_d	30.39	45.29	2.99	2.02	0.44
	2-2-2-8_e	27.72	71.67	0.18	0.00	0.10
	2-4-4-4_d	19.41	69.77	2.80	0.12	0.97
	2-4-4-4_e	23.53	76.46	0.01	0.00	0.00
	10-2-2-4_d	24.74	44.12	6.10	3.33	2.49
	10-2-2-4_e	27.17	64.81	2.78	0.86	0.41
	10-2-4-4_d	23.36	26.46	4.44	3.39	2.78
	10-2-4-4_e	26.49	48.41	4.07	0.63	0.81
	Average	25.35	55.87	2.92	1.29	1.00

5. #SAT-Solving

So far, this thesis has focused on improving the speed of the SAT-based test pattern generation through advanced modeling techniques and has looked into novel ways of solving the challenges that arise when generating test patterns for advanced fault models. The last two chapters of this work shift from advanced SAT-based modeling techniques for test pattern generation towards advanced techniques for circuit test in general. To this end, #SAT-solving is applied to circuit testing. This chapter introduces the distributed parallel #SAT solver dCountAntom while the subsequent Chapter 6 discusses the application of #SAT to the realm of circuit test.

As the name suggest, #SAT is tightly related to the SAT problem. In fact, a #SAT solver can be used as a SAT solver, and with slight modifications a SAT solver can work as a #SAT solver – albeit with sub-optimal performance. Therefore, much of the knowledge and experience from SAT-based ATPG can be transferred to #SAT applications.

In the author's master thesis, the thread-parallel #SAT solver countAntom was developed and presented. This chapter introduces the subsequent improvements that were added after the master thesis. They transform countAntom into the distributed parallel solver dCountAntom. Furthermore, both countAntom and dCountAntom were extended with a solve progress estimation that allows the user of the solver to gauge whether the current formula is solvable in the required time frame. This is a clear improvement over the classic timeout mechanism used by other solvers.

This chapter is structured as follows: In Section 5.1 an overview of the features of countAntom and of the general improvements used by state-of-the-art #SAT solvers is given. Thereafter, Section 5.2 introduces the distributed #SAT solver dCountAntom and the newly developed functions for the inter-process communication and synchronization. Section 5.3 presents the novel solve progress estimation and remaining solve time prediction. In Section 5.4 a thorough evaluation of dCountAntom and the newly introduced features is performed. Section 5.5 concludes the chapter with a summary.

This chapter is partially based on:

[C4] J. Burchard, T. Schubert, and B. Becker, “Laissez-faire caching for parallel #SAT solving”, in *SAT 2015*, ser. Lecture Notes in Computer Science, Springer International Publishing, 2015. DOI: `10.1007/978-3-319-24318-4_5`

[C5] J. Burchard, T. Schubert, and B. Becker, “Distributed parallel #SAT solving”, in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2016. DOI: `10.1109/CLUSTER.2016.20`

The main contributions by the author to this chapter are:

- The development of the distributed parallel #SAT solver `dCountAntom`.
- The development of a fast and efficient communication framework based on message passing.
- The development of different heuristics to decide when information is shared between different processes.
- The development of a solve progress estimation.
- The development of a total solve time prediction and a soft timeout mechanism based on this prediction for both `countAntom` and `dCountAntom`.
- The thorough analysis of the developed solver across a wide range of benchmark formulas.

The implementations are built on top of the #SAT solver `countAntom` [C4] which by itself is based on the SAT solver `antom` [13], [14].

5.1. countAntom

The #SAT solver `countAntom` is a thread-parallel (shared memory-based) #SAT solver. The parallelism of `countAntom` (and `dCountAntom`) is based on multiple solver threads or processes working on different nodes of the decision tree. The term node refers to a node in the decision tree which is annotated with its residual formula. In each node, a single decision variable is chosen. The assignment of this variable to *true* and *false* gives two new residual formulas and therefore two new nodes in the decision tree. These nodes are the children of the current node. Conversely, the current node is the parent of the newly generated nodes. Each node can also be considered to be a sub-problem of the overall model counting problem. For this reason, the terms node and sub-problem are used interchangeably in this chapter.

The general structure of `countAntom` follows that of a CDCL SAT solver as discussed in Chapter 2. At the beginning of the solve process, the formula is loaded and the root node of the decision tree is created and stored in a node manager. Next, the requested number of parallel threads is created and launched. At the core of every thread is a solve loop which, in each iteration, performs the required computations for a single node of the decision tree. The nodes are distributed among the threads by the node manager which also handles the synchronization. The solve loop is repeated until all nodes of the decision tree have been evaluated. Figure 5.1 shows the operations that are performed within the solve loop. These operations will be explained in detail in the subsequent sections.

While each thread performs its own computations on different nodes of the decision tree, the node manager is shared. The communication between the threads relies on a shared-memory concept where all of the threads can access the same memory, and is implemented using the boost threading library [76].

This section introduces the relevant features of `countAntom` that `dCountAntom` is based upon. For a full description of the original implementation the reader is referred to [C4].

Being based on the SAT solver `antom`, `countAntom` inherits many of the features of a modern SAT solver that were discussed in Chapter 2. These include conflict learning (lines 15 and 22) as well as a fast computation of the implications of the current assignment (lines 14 and 21). In addition, `countAntom` also contains #SAT specific improvements like sub-formula splitting (lines 17 and 24) and formula caching (lines 3, 8 and 11). Finally, to enable a parallel computation the general formula caching technique is extended into laissez-faire caching and an efficient thread synchronization is added.

Each of the #SAT specific improvements is discussed in the subsequent sections.

5.1.1. Sub-Formula Splitting

After the current decision variable was assigned and the implications of that assignment have been computed, it might be possible to divide the clauses of the residual formula into multiple sub-formulas which do not share any variables. The model count for each of these sub-formulas can then be computed separately and has to be multiplied to get the complete model count of the original node.

The new sub-formulas are considered to represent separate sibling nodes in the decision tree. Each new sibling node is added to the node manager. Hence, the siblings might be computed by different threads. Once the model count of the last sibling node was

```
n = NodeManager.getNextNode()
if Cache.contains(n) then
    mc(n) = Cache.getModelCount(n)
else
    moveToNodeInDecisionTree(n)
    if n is SAT then
        mc(n) = $2^{\#free\ variables}$
        Cache.addValue(n, mc(n))
    else if n is UNSAT then
        mc(n) =0
        Cache.nodeIsUNSAT(n)
    else
        v = decide()
        if assignAndComputeImplications(v, true) has CONFLICT then
            doConflictLearning($n$)
        else
            (pNode$_1$, ..., pNode$_i$) = split()
            NodeManager.addNodes(pNode$_1$, ..., pNode$_i$)
        end

        if assignAndComputeImplications(v, false) has CONFLICT then
            doConflictLearning($n$)
        else
            (nNode$_1$, ..., nNode$_j$) = split()
            NodeManager.addNodes(nNode$_1$, ..., nNode$_j$)
        end
    end
end
```

Figure 5.1.: The solve loop of `countAntom`.

computed, the overall model count of the original residual formula can be calculated by multiplying the model counts of all of the siblings.

As an example for sub-formula splitting, consider the formula

$$\Phi =(v_1 \lor v_2 \lor v_3) \land (\neg v_1 \lor v_2 \lor v_3) \land (v_1 \lor v_4 \lor v_5) \tag{5.1}$$

and the variable assignment $\pi = \{v_1 \to false\}$ which gives the residual formula

$$\Phi|_\pi =(v_2 \lor v_3) \land (v_4 \lor v_5). \tag{5.2}$$

The clauses $(v_2 \lor v_3)$ and $(v_4 \lor v_5)$ do not share any variables and $\Phi|_\pi$ can, thus, be split into two sub-formulas:

$$\Phi_1 =(v_2 \lor v_3) \tag{5.3}$$

$$\Phi_2 =(v_4 \lor v_5) \tag{5.4}$$

Since $mc(\Phi_1) = 3$ and $mc(\Phi_2) = 3$, the overall model count of $\Phi|_\pi$ is $3 \cdot 3 = 9$.

When a formula with n remaining variables can be split into i sub-formulas with $n_1, n_2, \ldots, n_i$ variables, the computational complexity is reduced from $\mathcal{O}(2^n)$ to $\mathcal{O}(2^{n_1}) + \mathcal{O}(2^{n_2}) + \cdots + \mathcal{O}(2^{n_i})$, which is always an improvement.

Furthermore, if even just a single one of the sub-formulas is unsatisfiable, the computation of the remaining sub-formulas can be aborted as the overall model count will be 0 anyway.

5.1.2. Formula Caching

During the solve process, many nodes in the decision tree might have the same residual formula. Instead of repeating the same computations for the same residual formula over and over again, a cache is utilized to store the model count of any node that has already been solved. If the exact same residual formula is encountered again, the result from the cache is used. This saves valuable computation time. For an efficient cache lookup, a hashing scheme is implemented to quickly compare nodes.

5.1.3. Laissez-Faire Caching

When combining formula splitting, caching and conflict learning, a problem arises [77]: To allow for an efficient utilization of the cache, conflict clauses are ignored when comparing the residual formulas.[1] Hence, even though the compared residual formulas might match, the residual conflict clauses might not.

Normally this would not cause a problem. Although conflict clauses restrict the solver's movements in the decision tree, they only crop unsatisfied areas and the number of satisfying solutions of a node should not be affected. However, when one of the sibling nodes of the current node is unsatisfiable this assumption is not valid any longer because the entire branch is unsatisfiable already. As was observed earlier, when a sibling of the current node is unsatisfiable, its model count becomes irrelevant. Therefore, even though the current node might be satisfiable, a conflict clause could still restrict some of the satisfying assignments and the node's model count might be incorrect. This is not a problem for the node itself – its model count will be multiplied with 0 anyway – but could yield an incorrect overall result if the node's model count is stored in the cache and the value in the cache is used for another node.

To resolve the conflict that arises from the combination of the three very important #SAT performance improvements, [77] simply removed the cached model counts of every node with an unsatisfiable sibling from the cache again. Since, in sequential #SAT solvers, the decision tree is traversed depth-first and in-order, this is a valid solution. It is, however, not applicable to a parallel solver where different threads might be working in completely different parts of the decision tree.

Laissez-faire caching solves this problem by storing a dependency every time a value from the cache is used. The cache itself can be utilized by all threads without any restrictions. Should a cached result be potentially incorrect, all nodes which depended on this result are informed and re-computed. In practice very few such invalidations occur on most formulas

[1] If conflict clauses would be considered for clause caching, the cache-hit rate would be drastically reduced since the solver constantly learns new conflict clauses and the residual formulas would, therefore, match less often.

and laissez-faire caching allows for a quick and efficient cache utilization by all parallel threads.

5.1.4. Thread Synchronization

When working with multiple threads, synchronization and data integrity are always a challenge. In `countAntom`, locks and mutexes – which ensure that only one thread can access a shared data structure at the same time – are used where necessary but as sparingly as possible to keep the overhead low. In addition to the cache, the different solver threads also share all conflict clauses to ensure that each thread always has access to the maximum amount of information. In comparison to a SAT solver, a #SAT solver derives fewer conflict clauses in the same time span because the advanced reasoning for sub-formula splitting and caching requires additional time. Therefore, it is possible to share all of the conflict clauses among the threads.

The initial formula loading is performed by a single thread that spawns additional threads which perform the calculations of the solve loop. Once there are no more nodes in the node manager, the threads are stopped and the initial thread returns the final model count that is stored in the root node of the decision tree.

5.2. dCountAntom

While `countAntom` successfully leverages the power of modern multi-core CPUs to increase the solve speed, it is limited by the number of cores that are available on a single machine since it relies on a shared memory communication approach. With `dCountAntom`, this restriction is removed by adding a message passing layer on top of the multi-threaded computations performed by `countAntom`. The resulting solver utilizes two different levels of parallelism: thread level parallelism based on shared-memory communication and distributed parallelism with message passing. By using shared memory communication for local CPU cores, the amount of messages that need to be transported is reduced and the performance is increased.

Figure 5.2 shows an overview of the general structure of `dCountAntom`. The solver is organized as a star network with a single process acting as the master that controls the computation. Each process can be seen as an instance of `countAntom` that is modified to communicate with other processes and to only compute the model count for smaller subproblems instead of the entire formula. The processes communicate by passing messages with the MPI framework [78]. This framework provides a high level view of the different processes and the communication infrastructure. Through MPI, messages can be transmitted over a multitude of different interfaces, including shared memory, Ethernet and InfiniBand. This is, however, completely transparent to `dCountAntom` which only needs to consider a simple send-and-receive interface.

5.2.1. Solve Flow

An overview of the overall solve flow of `dCountAntom` is shown in Figure 5.3.

The solve process of a Boolean formula Φ starts with the MPI agent spawning the requested number of parallel solve processes. From these processes, one is selected as the

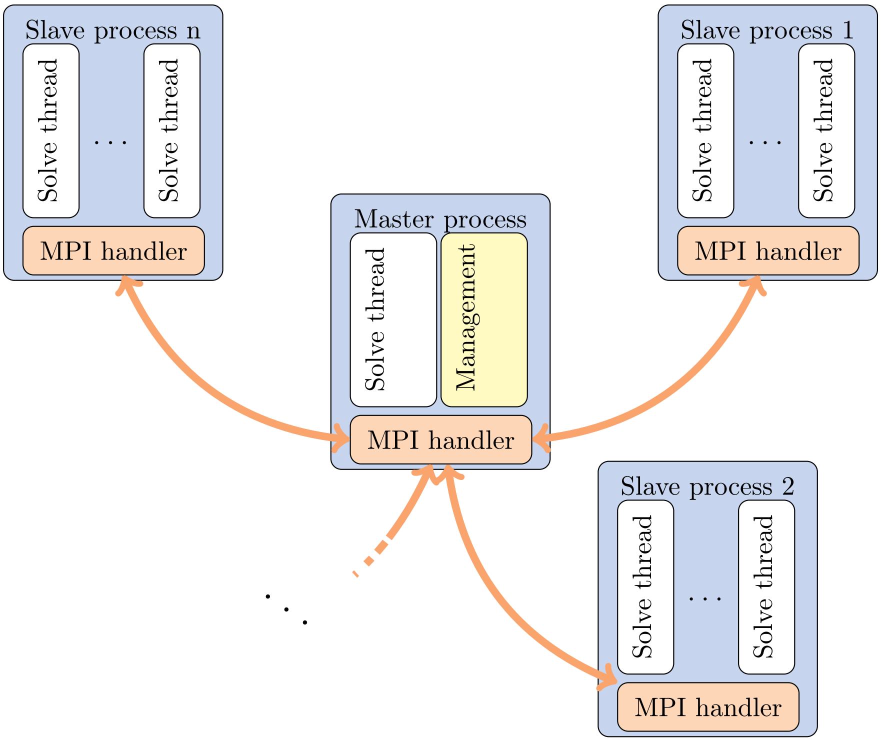


Figure 5.2.: The `dCountAntom` solver structure forms a star network.

master. All other processes are considered to be slaves. The master process then loads the formula Φ, performs some common preprocessing steps to simplify Φ and transmits the simplified formula to the slave processes. Next, the master splits the initial problem into multiple sub-problems by performing a certain number of decision steps. The sub-problems are then shared with the slave processes which solve them and return their model count. Since every slave process knows the entire formula the master only transmits the indices of the clauses in the sub-problem instead of the entire sub-formula. This greatly increases the efficiency of the communication. When new conflict clauses are found by the slave processes they are shared with the master which, in turn, shares them with all other slaves.

Once every sub-problem has been solved by the slaves, the master process computes the overall model count of Φ and returns it.

This basic approach is also utilized by some parallel SAT solvers (e.g., [79]). Here, of course, the slave processes do not return the model count of a sub-problem but only if it is satisfiable or not.

5.2.2. The Master Process

During the solve process the main task of the master process is the coordination of the slave processes to ensure an efficient computation. This section highlights the most important challenges of the master process, outlines how they are overcome and which adjustments can be made to optimize the computation. The master process itself is actually a fully

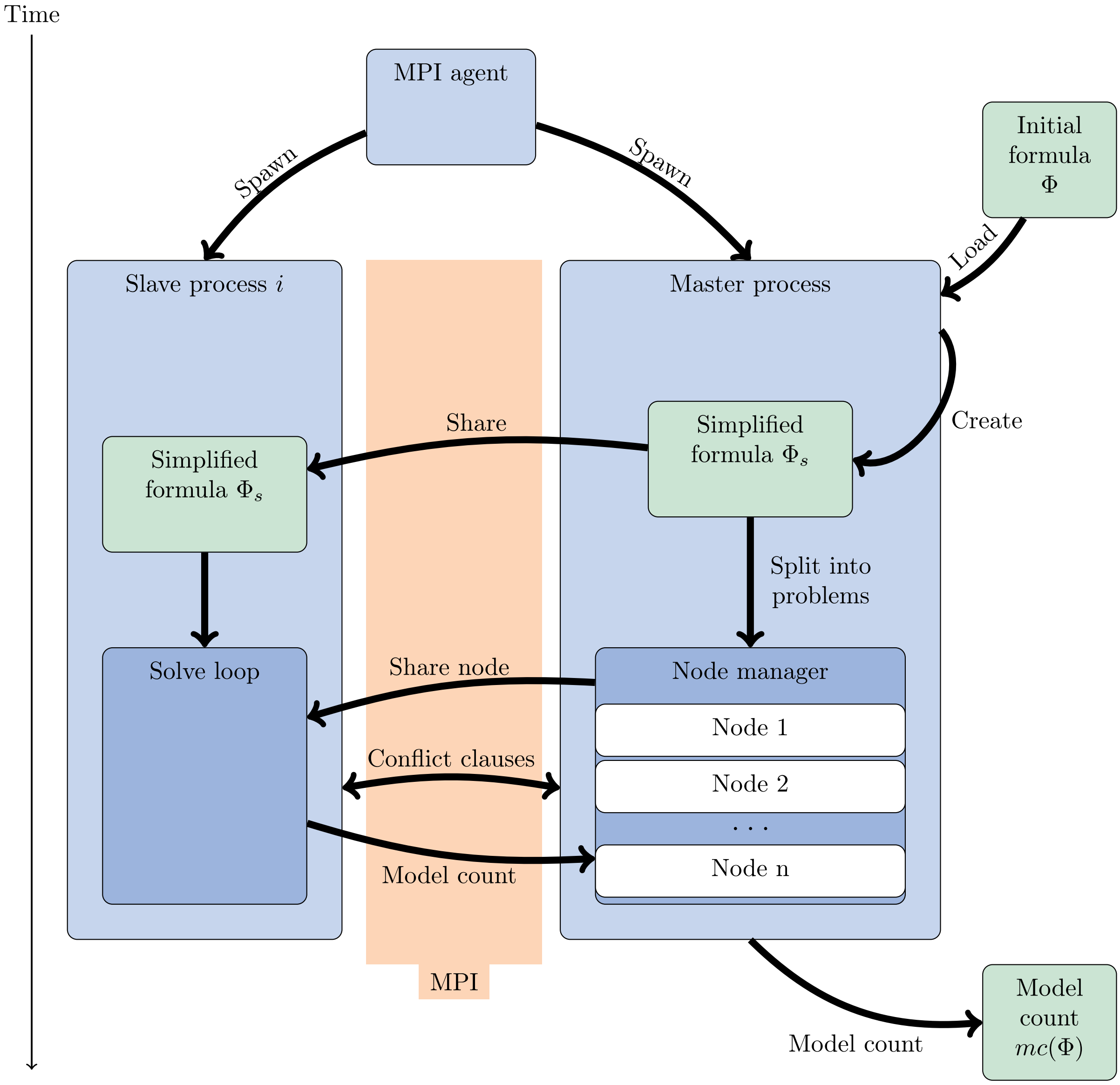


Figure 5.3.: `dCountAntom` solve flow. For simplicity only a single slave process is drawn.

functional single-threaded instance of `countAntom`, but is highly modified to provide the required functionality. Most importantly, instead of evaluating the decision tree all the way to a leaf, after a certain number of decisions it shares the nodes of its tree as sub-problems.

Creating the Sub-Problems

The first challenge lies with the splitting of the original problem into multiple smaller sub-problems which can then be transmitted to the slave processes. If the master creates too many, very easy to solve sub-problems, the communication overhead slows down the solver. On the other hand, if too few problems are generated, the workload might not be balanced equally among the individual slaves.

To allow for the optimization to certain problems, a user selectable variable, *target*, is introduced which describes the number of sub-problems that are to be generated by the master process. From *target*, the master computes the number of decisions that are performed before a node is shared with a slave process as $\lceil \log_2 target \rceil$. In addition to *target* the sharing of sub-problems is controlled by one more parameter, ν. A sub-problem is only shared when the corresponding formula contains at least ν variables.

Thus, while the parameter *target* controls the number of shared sub-problems, ν ensures that very simple problems are not shared at all. These problems are instead solved by the master process itself.

Load Balancing

The sub-problems that are generated by the master process often have very different difficulties. In the described parallel environment this becomes a problem as – in the worst case – a single slave process could spend a large amount of time solving a particularly hard node while the remainder of the problems have already been solved. In this case all other slaves are idle and the computation is not parallel anymore. To resolve this load balancing challenge a timeout mechanism is added: If a slave process cannot solve a problem within a user defined timeout (set to 5 seconds in the experiments), the node is returned to the master process. The master then performs at least 5 additional decisions which creates about 2^5 new nodes before the node's children are shared again. Due to caching and sub-formula splitting the actual number of new nodes that are generated by the master process varies.

Through this load balancing mechanism, difficult sub-problems are quickly split into smaller ones that can be handled by multiple slave processes again. This avoids a single process being stuck at a difficult problem for a very long time. The selection of the timeout has a large influence on the solve time. If the timeout is too low, many relatively simple sub-problems are unnecessarily split. When it is too large, the load balancing is limited.

Conflict Clauses

Learning from arising conflicts is one of the main pillars of fast modern SAT solvers. For a #SAT solver, cropping areas of the decision tree which are definitely unsatisfiable is of even higher importance since the solver will always traverse the entire decision tree. Thus, sharing the conflict clauses between the slave processes is a requirement for an efficient parallel #SAT solver. This sharing does not pose a problem in a shared memory architecture like that used by `countAntom` because it has very fast access and transfer times. However, it becomes more difficult when using relatively slow communication channels to transmit messages. Therefore, `dCountAntom` only shares conflict clauses up to a certain length. Since the amount of information that a conflict clause contains is inversely proportional to its length, this approach ensures that the most important knowledge is shared without creating too much overhead.

To avoid a large amount of point-to-point communication among the slaves, conflict clauses are first shared with the master process which collects the new conflict clauses of all slaves and distributes them in regular intervals.

5.2.3. The Slave Processes

The slave processes of `dCountAntom` are very similar to the `countAntom` solver. But instead of solving the entire formula Φ, the slave process only solves the currently assigned node and either returns its model count or cancels the computation if a timeout occurs. For an efficient computation it is very important that the slave processes are never idle. Therefore, each slave buffers a number of additional nodes, which should be computed after the current node, on a local queue. This ensures that the slaves are always occupied and do not have to wait for a new task from the master after the current sub-problem was solved. At the same time, the slave nodes should not store too many nodes since it would negatively impact the load balancing towards the end of the solve process. In addition, the sub-problems are generated on-the-fly by the master (only when a slave process has requested a new node) to incorporate all the knowledge of the conflict clauses during the node generation. Thus, it is beneficial to generate a new node as late as possible.

The generation of a new sub-problem by the master is triggered by the slave through a node request. Whenever the number of nodes on a slave's local sub-problem buffer is below a threshold, a new node is requested from the master.

Furthermore, the slave processes utilize sub-formula caching in combination with laissez-faire caching. Unlike conflict clauses, the cache information is not shared between the slaves for two reasons: Firstly, the amount of information stored in the cache cannot be transmitted through MPI without greatly reducing the overall solver performance. Secondly, the value of information in the cache seems to diminish quickly as the solver progresses further through the decision tree [77]. Thus, it can be assumed that the cached information of one slave process is not even helpful for another slave process that is working at a completely different position in the decision tree.

5.2.4. MPI Communication

At the heart of `dCountAntom` lies the inter-process communication through the MPI framework. This is performed by an MPI handler in each solve process. The handler is responsible for crafting and sending as well as receiving and decoding the different messages.

In total, only eight different messages are required for the communication between the master and the slaves. In MPI there are two different classes of messages: Broadcasts are transmitted to all other processes. Unicasts are used for point-to-point communication. A unicast message can be transmitted and received in the background, whereas a broadcast can only be transmitted once all processes are waiting to receive it. Since the slave processes are not synchronized and act independently, `dCountAntom` generally utilizes nonblocking unicast communication to avoid blocking the slaves while waiting for broadcasts. The following list summarizes the different messages that are being transmitted:

- **Initial formula**: The master transmits the initial formula to all of the slaves as a broadcast. Since the slaves have not started the computation yet, the synchronization is not a problem here. This is the only broadcast during the whole solving process.
- **Node request**: A slave requests a new sub-problem from the master. Every node request will at some point be answered by the master – either with a sub-problem or by signaling that the entire formula was solved. Note that the slave process can continue to work on the next node from its local queue already while waiting for a response from the master.
- **Node**: The master transmits a new sub-problem to a slave. To reduce the message size, the master does not transmit the actual sub-formula that corresponds to the sub-problem. Instead, it sends the indices of the clauses in the original formula that are still present in the sub-formula in combination with the remaining unassigned variables.
- **Conflict clause**: A single conflict clause or multiple conflict clauses that are grouped together to reduce the number of messages.
- **Model count**: A slave process returns the model count of the current sub-problem to the master.
- **Timeout**: A slave process informs the master that it could not solve the assigned sub-problem within the available time frame. The master will then further split the problem while the slave process continues with the next node on its local queue.
- **Statistics**: Solve statistics (e.g., number of solved nodes, cache hit rate, ...) from the slave processes are transmitted to the master which aggregates the information and presents it to the user.
- **Computation finished**: A signal from the master that indicates that the entire formula was solved. This triggers the transmission of the solve statistics by the slaves. Afterwards, the slaves are shut down as they are no longer needed.

Based on these messages, the communication of `dCountAntom` is lightweight and non-intrusive and the slave processes can act with a large amount of independence.

5.3. Progress Estimation

The inherent difficulty of the #SAT problem means that many formulas are very difficult to solve. Thus, the user of a #SAT solver is often left wondering whether the solver will need "just another minute" to return a solution or is practically stuck and might still run for a very long time.

In established SAT or #SAT solvers, there is very little indication with regard to the solve progress after starting the solver. In most use cases, the solver is started with a timeout and either solves the formula within this timeout or skips it. This approach was, for example, used in the stuck-at and TSOF ATPGs that were presented in the previous chapters.

To give more information with regard to the progress of the solver, a solve progress estimation was developed and implemented into `countAntom` and `dCountAntom`. The solve progress estimate is then used to predict the overall solve time as well as the remaining solve time.

More insight into the predicted solve time allows for better decisions by its user. The classical timeout mechanism defines a hard deadline after exactly X seconds. However, for many applications it is desirable to spend a little bit more time to find a solution for a problem when it can be found in $X + Y$ seconds. This effect can of course be achieved by setting the overall timeout to $X + Y$. However, for other problems that cannot be solved neither in X nor in $X + Y$ seconds, this would waste even more time. The proposed solution utilizes the predicted solve time to enforce a soft timeout and a hard timeout during the solving of the formula.

Independent of the estimated solve progress, the solver will always continue the calculation as long as the soft timeout (X) is not reached. After the soft timeout is reached, the calculation is continued as long as the predicted total solve time is below the hard timeout ($X + Y$). Otherwise it is aborted. Therefore, the solver will stop the calculations after the soft timeout if it is unlikely that the formula is solvable in a reasonable time. The reasonable time limit is used as the hard timeout.

To the knowledge of the author, no other SAT or #SAT solver offers this kind of additional information and control to the user of the solver. The combination of the soft and hard timeout allows the solver to compute a solution for formulas when it is in reach and aborts the computations early when it is not.

The gains of utilizing the soft timeout mechanism are highlighted further in Section 6.4.4 in the subsequent chapter.

5.3.1. Estimating the Solve Progress

The progress of the solver is estimated by calculating the fraction of the decision tree that has already been covered by the solver. This fraction is calculated by traversing the entire decision tree and counting the share of finished nodes for which the model count is already known. When computing the fraction, $frac(n)$, of the decision tree that is represented by a node n, sub-formula splitting has to be considered as it often creates many sibling nodes representing different portions of the residual formula.

The calculation of the share of the decision tree that is already finished is performed by traversing the decision tree depth first. If a node n is finished, $frac(n)$ is added to the overall solve estimate. Otherwise, the fractions of the children nodes are computed. To this end, $frac(n)$ is first split evenly between the positive and negative branch. Next, each child node n_c in the two branches is assigned a value $frac(n_c)$ that is proportional to the number of variables that the node represents in comparison to the total number of variables in the residual formulas.

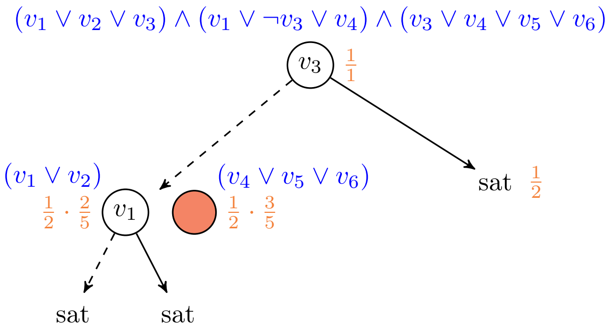

$(v_1 \lor v_2 \lor v_3) \land (v_1 \lor \neg v_3 \lor v_4) \land (v_3 \lor v_4 \lor v_5 \lor v_6)$
v_3
$\frac{1}{1}$
$(v_1 \lor v_2)$
$\frac{1}{2} \cdot \frac{2}{5}$
v_1
$(v_4 \lor v_5 \lor v_6)$
$\frac{1}{2} \cdot \frac{3}{5}$
sat
$\frac{1}{2}$
sat
sat

Figure 5.4.: Estimating the solve progress of the #SAT solver by counting the fraction of the decision tree that has already been analyzed. Each node is annotated with the fraction it represents.

Figure 5.4 shows an example of the fractions in a partially evaluated decision tree. In the example, the solver has already computed the model count for most nodes. Only the node marked in red has not been evaluated, yet and is still missing. Each node in the tree is marked with the fraction of the overall problem that it represents. Note that after assigning v_3 to $false$, the residual formula $(v_1 \lor v_2) \land (v_4 \lor v_5 \lor v_6)$ is split into two sub-formulas. The first sub-formula represents $\frac{2}{5}$ of the remaining variables, the second one $\frac{3}{5}$ of the remaining variables.

To compute the overall solve progress estimate the values of all the finished parts is summed up. Thus, the overall solve progress is estimated as:

$$\frac{1}{2} + \frac{1}{2} \cdot \frac{2}{5} = \frac{7}{10} = 70\,\%. \tag{5.5}$$

To quickly compute the progress estimate, the number of nodes in the tree should be as low as possible. In `countAntom`, the optimal time for a progress estimation is after a cache cleanup operation, which is regularly performed and removes all finished nodes from the tree. In `dCountAntom` the progress estimation is only performed by the master since the number of nodes in the master is low anyway. Here, the estimate is updated every time a slave transmits the model count of another sub-problem to the master.

It should be noted that the developed progress estimation mechanism could in theory also be applied to SAT-solving. However, for satisfiable formulas the SAT solver does not traverse the entire decision tree (because it only searches for a single satisfying solution). Therefore, the progress estimate based on the traversed share of the decision would potentially be highly inaccurate and of far less value. A #SAT solver on the other hand always has to traverse the entire decision tree, enabling the presented approach.

5.3.2. Predicting the Solve Time

Based on the previously computed solve progress estimate, the overall as well as the remaining solve time can be predicted. To this end, the solver first computes the change of the progress per time unit, $\frac{d\ progress}{dt}$, which is then used to calculate a prediction for the time that will still be required to cover the part of the decision tree that has not been considered by the solver, yet.

Since the #SAT problem is very difficult to solve and #SAT solvers rely heavily on heuristics to make "smart" decisions, the solve progress estimate is not guaranteed to be accurate. In fact, for some formulas the progress estimate quickly jumps to a relatively large value but then does not progress any further. This might occur because the solver can quickly mark a large part of the decision tree as satisfiable or unsatisfiable but other parts of the tree are highly complex and require a large number of computations. To even out such jumps in the progress estimate, the progress per time is averaged with the previous estimates.

5.4. Evaluation

The presented distributed parallel #SAT solver `dCountAntom` is evaluated on a cluster with up to 256 CPU cores used in parallel. The author acknowledges the support of the state of Baden-Württemberg through bwHPC, which provides access to the computation cluster used for the experiments.

For a thorough evaluation of `dCountAntom`, benchmark formulas were sourced from three different circuit-based #SAT applications:

OP: The output probability (OP) formulas encode the probability that the output of a circuit is '1' when the circuit inputs are chosen freely. While these benchmarks focus on the circuit's output, it is generally possible to accurately calculate the '1'-probability of any signal in a circuit through #SAT.

FI: The fault injection (FI) formulas encode the probability that a fault is successfully injected into a flip-flop by an adversary that can shorten the length of the circuit's clock period [W1].

PD: The possibly detected (PD) formulas encode the probability of detecting a possibly detected fault. This #SAT application is discussed in detail in the subsequent Chapter 6.

Section 5.4.1 shows the single-threaded performance of modern #SAT solvers to put the presented results into perspective. Thereafter, Section 5.4.2 evaluates the speedup that `dCountAntom` obtains by utilizing many CPU cores in parallel.

5.4.1. Single-Threaded Performance Comparison

Unlike SAT-solving where many different solvers that are constantly improved exist, there are only a few different #SAT solvers. For the comparison of the single-threaded performance, the latest accurate (`sharpSAT` [80]) as well as approximating (`ApproxMC` [81], [82]) solvers are analyzed. A comparison to earlier #SAT solvers was performed in [C4] and showed that `sharpSAT` usually outperforms them by a wide margin.

The distributed solver `dCountAntom` itself does not offer a single-threaded mode. Therefore, `countAntom` is used for the following comparisons. Table 5.1 gives an overview of the different benchmark formulas and the solve times of `countAntom`, `sharpSAT` and `ApproxMC` (with a model count tolerance of ±10 % and a confidence of 95 %). Entries marked with X indicate a timeout after 72 h. Since the single-threaded runtime of `countAntom` is used to compute the speedup with multiple CPU cores in the next experiments, each formula is solved 10 times and the results are averaged to avoid random influences.

Table 5.1.: The number of variables and clauses in each of the benchmark formulas and the single core solve time of `countAntom`, `sharpSAT` and `ApproxMC` in seconds.

Formula	#Variables	#Clauses	`countAntom`	`sharpSAT`	`ApproxMC`
OP_b14c	1 172	3 194	**2 922.7**	4 911.8	X
OP_c1355	349	936	**429.6**	X	X
OP_c5315	622	1 809	**19 233.3**	X	X
OP_c6288	877	2 546	5 825.0	**1 273.8**	X
FI_33_20	1 040	3 862	**7 762.7**	X	X
FI_33_18	1 040	3 860	**11 409.6**	X	X
FI_33_16	1 040	3 858	**30 837.2**	X	X
FI_33_14	1 040	3 856	**145 155.5**	X	X
PD_p35k	10 159	26 771	1 135.4	**82.3**	X
PD_p100k	709	2 163	1 970.6	**1 355.1**	X
PD_p388k	1 867	5 533	**843.8**	2 795.8	X
PD_p533k	1 187	3 512	1 701.6	**1 245.0**	X

The single-threaded performance can be summarized into two main observations:

Firstly, the performance of a #SAT solver is formula-dependent. Generally, `countAntom` is the fastest solver, but on some formulas it is outperformed by `sharpSAT`. Nonetheless, `countAntom` is clearly a competitive solver even with just a single CPU core.

Secondly, current approximating solvers do not offer the required performance for calculations with a high accuracy and confidence. Of course, the performance of `ApproxMC` could be improved by accepting a higher tolerance or lowering the necessary confidence. This could, however, greatly reduce the quality of the results which – depending on the application – is not acceptable.

5.4.2. Distributed Parallel Performance

To evaluate the distributed performance of `dCountAntom`, the solve time with 16, 32, 64, 128, 192 and 256 CPU cores is analyzed. Each process is allocated 4 CPU cores on the same machine and utilizes thread level parallelism through shared memory. Therefore, between 4 and 64 different processes are used for the experiments. One of these processes becomes the master, the remaining processes are used as slaves.

For the experiments the number of nodes that are to be created by the master process is fixed to 200 (*target*) and nodes with less than 30 variables are not shared at all (ν) but solved by the master directly. Figure 5.5 shows the obtained speedup compared to the single thread runtime of `countAntom` (shown in Table 5.1) for the different formulas in detail.

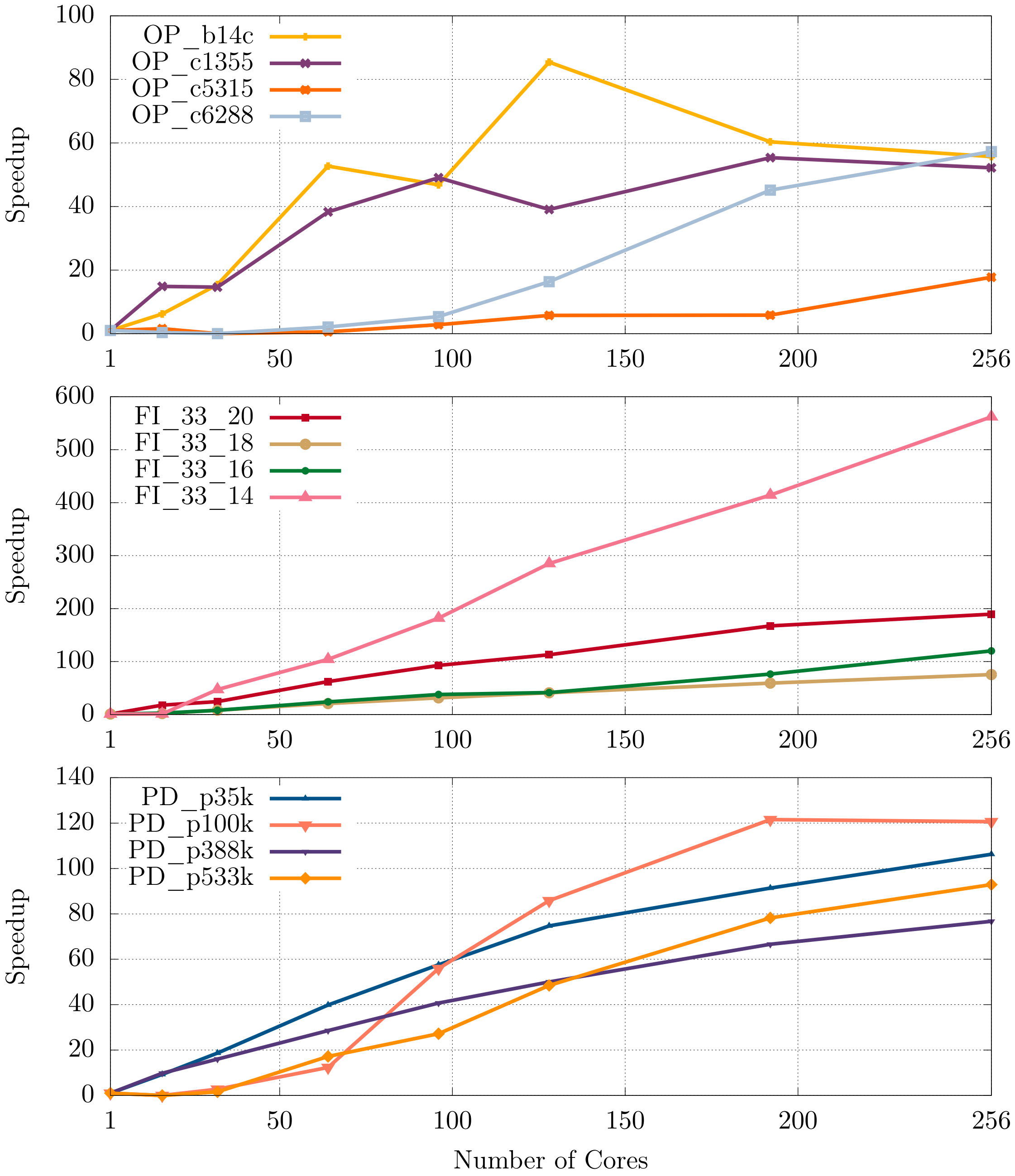


Figure 5.5.: **Speedup** of `dCountAntom` with up to 256 CPU cores compared to the single thread runtime of `countAntom`.

For all of the formulas, dCountAntom achieves significant speedups. In the most successful case, for the formula FI_33_14, the solver is about 562 times faster when utilizing 256 cores than it is with a single core. This hyperlinear speedup can be explained through the changed computation order. With more parallel solvers, the nodes of the decision tree are evaluated in a different order which leads to different decisions and conflict clauses. These minor differences can have a great impact on the overall performance of the solver – for better or for worse.

The influence of the order of computation on the solve time also explains the fluctuations for different numbers of cores that can be observed on some formulas. Sometimes, adding more CPU cores might actually slightly increase the solve time. Consider for example the formula OP_c1355. When solving the formula with 96 cores, the speedup is larger than that with 128 cores. Nonetheless, a clear overall trend can be observed for all analyzed formulas: With more CPU cores a higher speedup is obtained.

On average, the solver is about 127 times faster with 256 cores. Furthermore, for many formulas the slope of the speedup does not appear to decrease and it can be assumed that adding more cores would increase the speedup even further. However, for some other formulas (e.g., PD_p100k) the speedup with 192 and 256 cores is basically identical. This lack of additional speedup can be explained by analyzing the absolute solve time. With 192 cores, the formula PD_p100k can be solved in 16.2 s and with 256 cores in 16.3 s. With such short solve times, the overhead of generating many parallel solvers and transmitting messages between them becomes larger than the gain of the additional cores.

In comparison to SAT-solving where parallel solving has been attempted with mixed results [79], [83]–[88], dCountAntom achieves a speedup for every single analyzed formula. Some parallel SAT solvers (e.g., [86]–[88]) do not even attempt to solve the problem cooperatively but instead start multiple instances of the same solver with different parameters with the aim of providing a fast result through the different ways the search space is covered. In #SAT-solving, a more structural approach with solve processes that work together to compute the overall model count is beneficial, since the entire decision tree has to be covered anyway.

Number of Solved Nodes

The variables *target* and ν are used to control the number of initially generated sub-problems. However, the load balancing mechanism allows the master to further split particularly challenging sub-problems. Figure 5.6 shows the number of sub-problems that are actually being solved by the slave processes. The gray markings at the very bottom of the graph indicate the target of 200 nodes. Clearly, the target is exceeded for every single formula, often by a wide margin (note that the y axis is scaled logarithmically). The load balancing mechanism is obviously splitting many sub-problem into additional smaller sub-problems. However, a large number of solved nodes is not always an indicator for a large overall solve time or low speedup. Consider for example the formula FI_33_14, which has the largest overall speedup with 256 cores even though about 18 000 sub-problems were solved.

The number of solved sub-problems differs depending on the number of slave processes. The reason for the different number of solved sub-problems is, again, the changed order of computation when utilizing additional solve processes. The results show that the computation order can result in up to 10 times more sub-problems that have to be solved by `dCountAntom` for some formulas. Consider, for example, the number of solved sub-problems for the formula OP_c5315 (shown as the top orange line in Figure 5.6): With 16 cores about 54 000 sub-problems are solved. On the other hand, with 96 cores, about 502 000 sub-problems have to be considered.

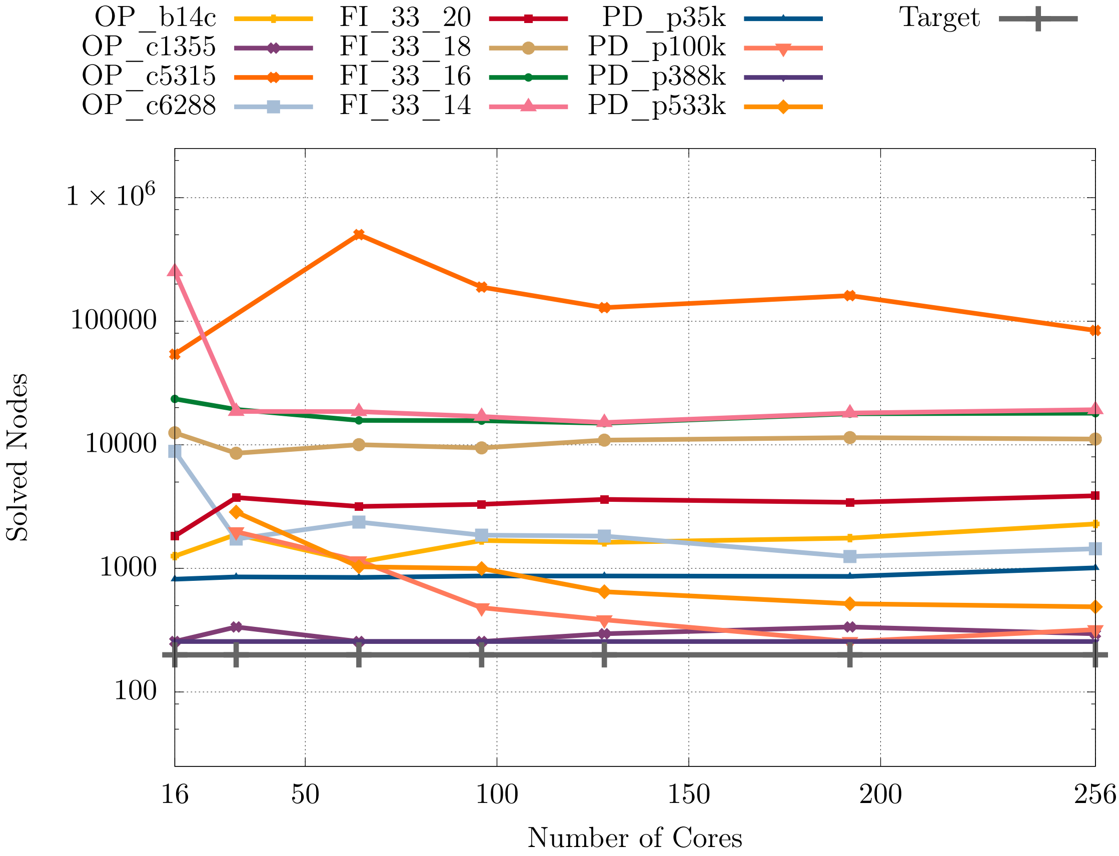


Figure 5.6.: **Number of solved sub-problems** by the slave processes.

Timeouts

The load balancing of `dCountAntom` relies on timeouts to split hard problems into smaller ones to ensure that all slave processes remain occupied throughout the solve process. As the previous section revealed, in `dCountAntom` many additional smaller sub-problems are created. In this section, the occurring timeouts are further analyzed by considering the solve process of the formula OP_c6288 in detail.

In Figure 5.7, all of the timeouts that occur during the solve process of OP_c6288 are visualized. Most timeouts occur at the beginning of the solve process after the splitting of the initial problem into sub-problems. Afterwards, the number of timeouts usually stagnates. Only when utilizing 16 cores, timeouts still regularly occur and the total number of timeouts is much larger.

The total number of timeouts differs depending on the number of CPU cores. This can, again, be explained by the different computation order when a different number of slave processes is used. The process of solving the #SAT problem in `dCountAntom` is driven by heuristics. With a different number of slave processes, the evaluation order of the decision tree changes and other conflict clauses could be learned. Thus, some timeouts might be avoided because additional knowledge is available.

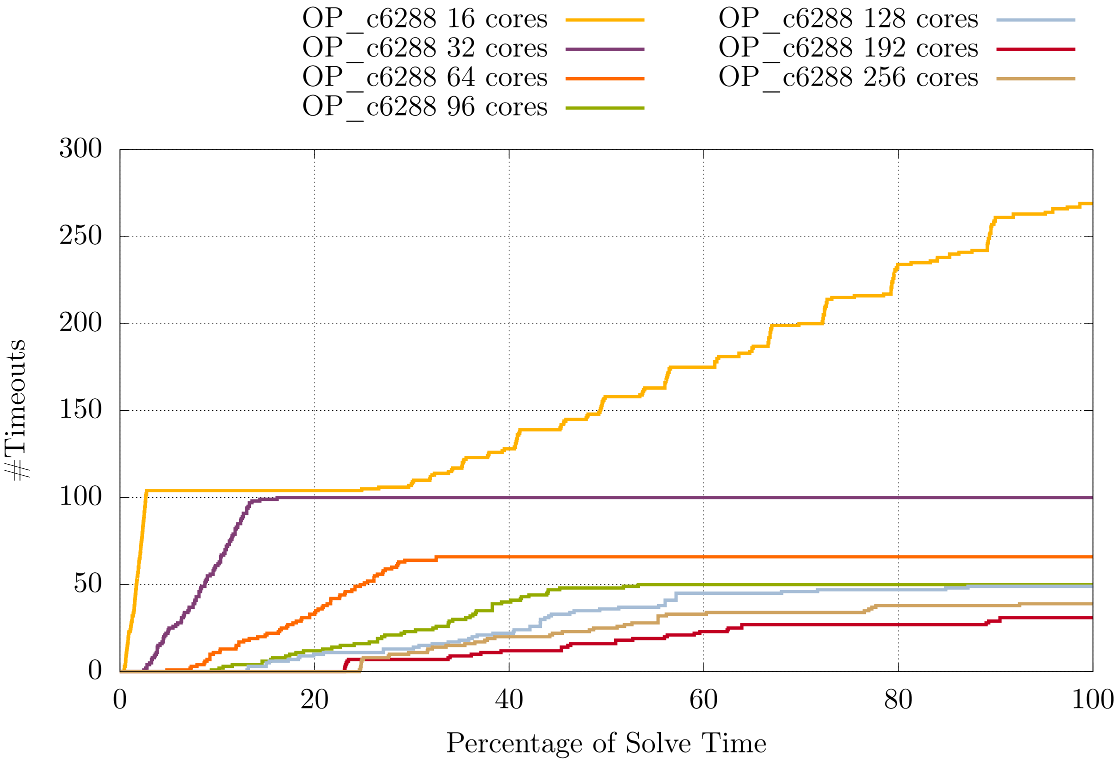


Figure 5.7.: **Cumulated number of timeouts** while solving *OP_c6288* with different numbers of CPU cores.

Of course, a large number of timeouts is not desirable as every timeout corresponds to CPU time spent to no avail without actually progressing in the calculations. However, predicting the difficulty of a sub-problem before actually solving it – which would allow the solver to further split the node right away instead of encountering a timeout – is difficult due to the nature of the #SAT problem itself. Of the numerous different approaches for load balancing and difficulty prediction that were analyzed, the presented timeout mechanism introduces the smallest overhead and fastest overall speedup. As the results clearly show, `dCountAntom` delivers great speedups with many CPU cores even though timeouts occur.

5.4.3. Solve Progress Estimation

The estimation of the solve progress and prediction of the overall as well as remaining solve time are important new features to establish #SAT-solving as an applicable solving technique for very difficult problems. Given the high complexity of the problem class, it is not surprising that the solve time for some formulas is unacceptably high.

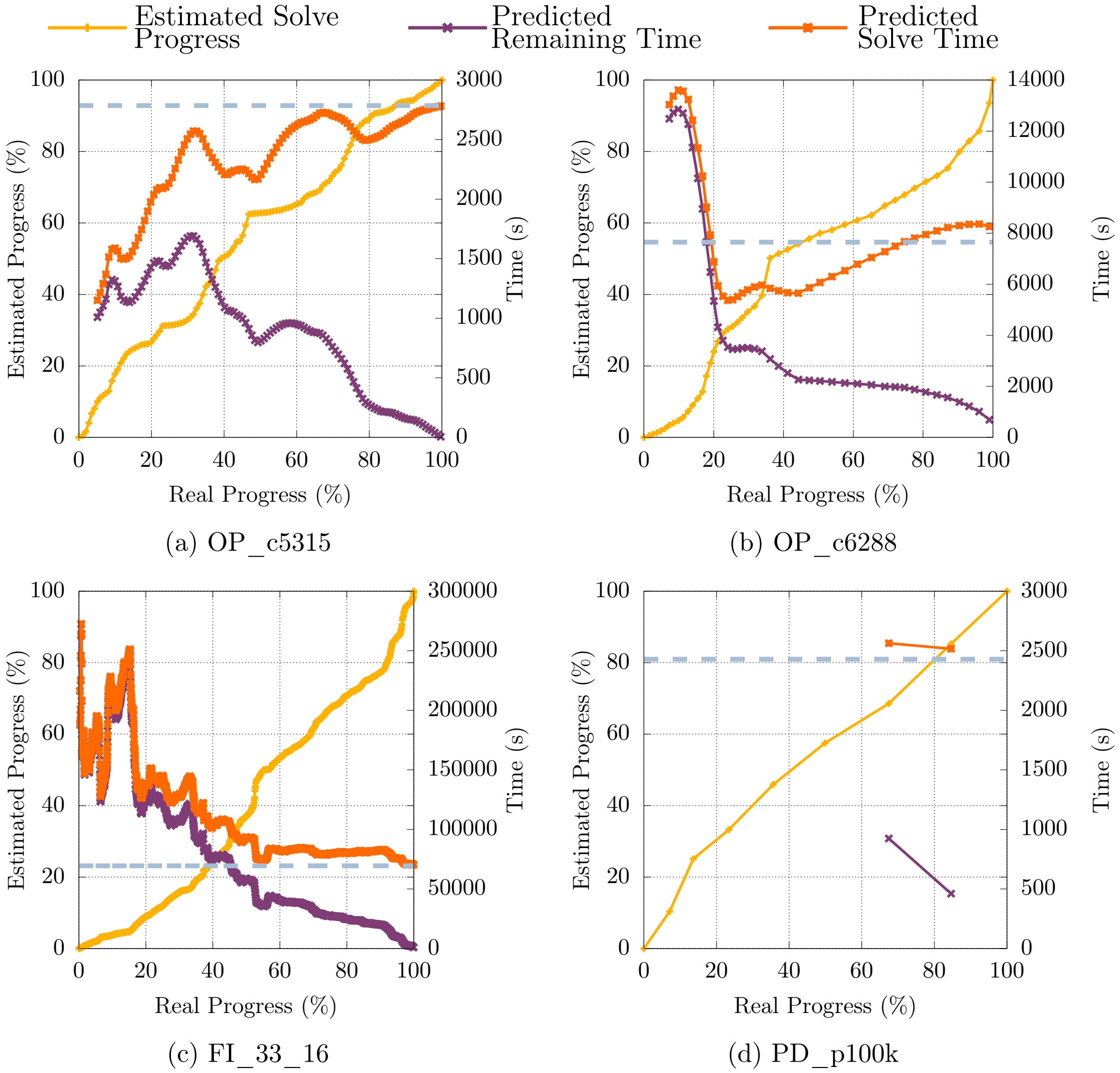
Based on the predicted solve time, a soft timeout mechanism can be used to abort the calculation on formulas which cannot be solved in a very long time. This ensures that the solver does not spend too much time on these formulas. To evaluate the progress estimation, the solve progress is analyzed more closely with `countAntom`. The solve progress estimation in `dCountAntom` works exactly the same way. However, due to the longer solve time the results of `countAntom` with only one thread are more demonstrative.

Figure 5.8 shows the progress estimate as well as predicted solve time for four different formulas in comparison to the real solve progress and the real solve time. The real solve progress is calculated by dividing the solve time up to the considered point in time by the total solve time. A perfect prediction would show the solve progress estimation (yellow line) rise from 0 to 100 linearly and a constant predicted solve time (orange line).

The solve progress estimate (yellow line) is relatively accurate for all four different analyzed formulas. The prediction of the total solve time (orange) as well as the remaining time (violet) is not as accurate, especially in the beginning of the solve process. Nonetheless, the predictions are still beneficial to gauge the general difficulty of the formula. For this it is not relevant if the final solve time is twice as long or short as initially predicted. Instead, the prediction allows the user to understand the general difficulty and whether the solving time will be in the range of minutes, hours, days – or even longer.

The prediction of the final solve time is only computed after the first 5 solve progress estimates have been finished. This makes the prediction more stable because initial jumps in the estimated progress are ignored. As a result, for formulas that are quickly solved the predicted solve time is only available relatively late. In the examples, this is the case for PD_p100k.

Overall, for all four analyzed formulas the total solve time prediction is in the same range as the real solve time and can definitely be used as an advantageous indication.



(a) OP_c5315

(b) OP_c6288

(c) FI_33_16

(d) PD_p100k

Figure 5.8.: **Progress estimate** and **predicted remaining and total solve time** for four different formulas. The dashed blue line shows the real solve time.

5.5. Summary

This chapter introduced and discussed the distributed parallel #SAT solver dCountAntom which is designed to solve difficult #SAT problems through a divide and conquer approach utilizing many CPU cores across different machines at the same time. Furthermore, a method to estimate the solve progress and to predict the total solve time were presented. With this additional information the user of the solver can make more informed decisions with regard to aborting a solve process when it seems unlikely that the problem can be solved in the required time.

The key points of the chapter are the following:

- This work introduced the first distributed parallel #SAT solver dCountAntom which can efficiently solve the #SAT problem with many CPU cores. The evaluation shows that with 256 CPU cores the solver is on average about 127 times faster than on a single core.
- Special care has to be taken to make sure that all slave processes are always supplied with work. To this end, a load balancing mechanism based on timeouts is utilized. This ensures, that difficult-to-solve sub-problems are further split and the workload can be spread across the different CPUs.
- Through a solve progress estimation the remaining and total solve time can be predicted. This prediction allows the user of the solver to make an informed decision about whether to abort the solve process or whether to wait for it to finish. The progress estimate is furthermore used for a novel soft timeout mechanism.

Since #SAT-solving is a relatively new field of study, there are many possibly starting points for future work on (parallel) #SAT-solving. These include the development of new and advanced heuristics for the solve process and the load balancing. Furthermore, for some applications an estimate of the model count of a formula might be sufficient. However, the current approximating solvers perform poorly on the circuit-derived formulas that are considered in this thesis and do not offer an alternative, yet.

Overall, the presented distributed #SAT solver dCountAntom clearly highlights the potential of utilizing grid-scale parallelism when solving the #SAT problem. This is especially relevant since the #SAT problem is inherently difficult to solve and previous solvers might be too slow for many real life problems.

6. Accurate Characterization of Possibly Detected Faults

This work clearly showed that utilizing a SAT-based modeling in the realm of circuit test comes with many benefits. These range from the fast computation times for difficult problems over the continued speed improvements through better and better solvers to the simplicity of modeling even highly complex requirements and effects into the Boolean formula.

Nonetheless, the basic SAT problem only provides yes or no answers – the formula is either satisfiable or it is not. While this is sufficient for many problems, for some applications a simple binary answer does not suffice. Instead, the probability of an event to occur could be relevant. Such a probability might be computable by mapping the problem to a Boolean formula and solving this formula through a #SAT solver.

In this chapter, a novel, #SAT-based approach for the accurate computation of the detection probability of possibly detected faults is introduced. When the detection of a fault depends on unknown or undefined circuit inputs and the ATPG algorithm cannot prove that it is detected but also cannot prove that it is definitely not detected, the fault is classified as possibly detected. By accurately computing the probability that the fault is detected, the overall fault coverage can be computed with a higher accuracy. Furthermore, this information can be used as the basis for testability improvements, for example the insertion of test points for faults with a low detection probability.

This chapter is structured as follows: Section 6.1 introduces the concept of possibly detected faults. Thereafter, Section 6.2 discusses the accurate #SAT-based algorithm to characterize possibly detected faults by computing their detection probability. In Section 6.3, four different improvements to increase the solving speed are introduced. The presented algorithm is extensively evaluated in Section 6.4. Section 6.5 concludes the chapter with a summary.

This chapter is partially based on:

[C6] J. Burchard, D. Erb, and B. Becker, “Characterization of possibly detected faults by accurately computing their detection probability”, in *Design, Automation and Test in Europe (DATE), 2018*, 2018. DOI: 10.23919/DATE.2018.8342040

The main contributions by the author to this chapter are:

- The automatic generation and extraction of the list of possibly detected faults from a commercial ATPG tool.
- The development and implementation of the algorithm to accurately compute the detection probability of possibly detected faults.
- The development and implementation of four different SAT-based improvements to further increase the speed of the computations by simplifying the problem.
- The thorough evaluation of the presented approach with possibly detected faults in a large variety of different circuits.

The implementations are built on top of the PHAETON framework [44] by Matthias Sauer which, among others, provides a circuit file parser and logic generators for the Tseitin transformation.

6.1. Possibly Detected Faults

For many faults it is sufficient to specify some of the inputs of a circuit to ensure the activation of the fault and the propagation of its effect to an output. The remaining inputs are irrelevant for this particular fault. In the corresponding test pattern, these unspecified inputs can be marked with an X as they can be assigned to both ‘0’ and ‘1’ without changing the outcome of the test. The existence of X values gives rise to techniques like test compaction [89] and test compression [90] which rely on unspecified inputs.

In addition to unspecified inputs, there might also be inputs that are unknown. The value of a circuit input might be unknown for different reasons:

- The circuit input is connected to a flip-flop that is not part of a scan-chain and has no set or reset functionality. It might be possible to initialize this flip-flop through an initialization sequence [91], [92]. However, such a sequence might require many clock cycles or might not exist at all [93].
- The circuit input is connected to another on- or off-chip device that cannot be controlled during the test.
- The circuit input is connected to an analog partition of the chip that cannot be controlled during the test.
- The circuit input is connected to a device or module that is not fully specified yet.

For the purpose of possibly detected faults unknown circuit inputs are modeled similar to unspecified inputs: with the value X. The exact reason for an X occurrence at an input is irrelevant for the remainder of this chapter.

Just like a normal input value, X values can be propagated through the circuit. The handling of X values during the test pattern generation [45], [46], [94] and fault simulation [95], [96] has been considered from different angles which include, among others, accurate but costly quantified Boolean formula (QBF) based ATPG and simulation. The difficulty in accurately considering X values arises from the fact that re-converging X values might disappear and become ‘0’ or ‘1’ again [94].

A test pattern possibly detects a fault at a circuit output if the output has a specific (non-X) value in the good circuit but is X in the bad circuit. Thus, depending on the actual values of the X-inputs when applying the test to the device, the fault might be detected or not. Figure 6.1 shows the effects of the test pattern “1X0” for two different stuck-at faults.

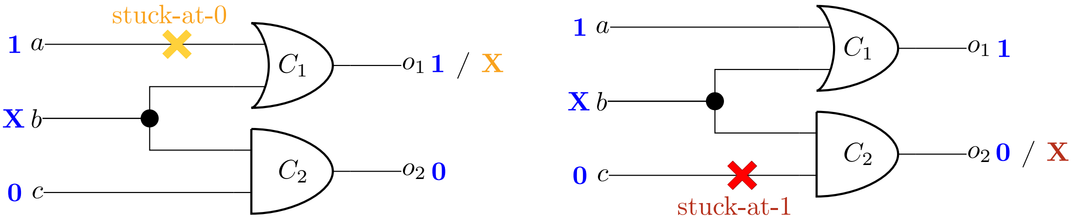


Figure 6.1.: The test pattern “1X0” possibly detects both the stuck-at-0 fault as well as the stuck-at-1 fault.

When input b is assigned to ‘0’ the test pattern successfully detects the stuck-at-0 fault at the first input of C_1 but not the stuck-at-1 fault at the second input of C_2. When b is assigned to ‘1’ the situation is reversed. Hence, both faults are detected with a probability of 50 % when b is freely assigned.

It should be noted that commercial ATPG tools usually count a share of the possibly detected faults towards the overall fault coverage. The share is controlled through a user selectable factor [97], [98]. Thus, the tool assumes that a certain fraction of possibly detected faults will also be detected in a real circuit.

The goal of the presented approach is an accurate characterization of the possibly detected faults by computing the actual probability that a possibly detected fault is really detected when the X-inputs are chosen freely. In this chapter, the focus lies on possibly detected stuck-at faults. However, being a SAT and #SAT-based approach, other fault models could be supported with ease.

6.1.1. Classification of Test Patterns

The characterization of possibly detected faults is based on a test pattern set. This set is created through a commercial ATPG tool, which is used to generate test patterns for all stuck-at faults within a circuit. The ATPG classifies each fault into one of three categories:

- **Detected**: The fault is detected by a test pattern.
- **Untestable**: The ATPG has proven that (under the current constraints) no test pattern can detect the fault.
- **Possibly Detected**: The ATPG found one or multiple test pattern(s) that might detect the fault if the X-inputs are assigned correctly.

For the analysis the list of faults that are only possibly detected is extracted. Furthermore, for each possibly detected fault the test patterns which might detect the fault are stored. The outcome of this first step is a file containing a list of possibly detected faults and the test patterns which possibly detect each fault. This file is utilized by the proposed algorithm that is discussed in the next section.

To simulate the test pattern generation for circuits with unknown inputs the aforementioned process with the commercial ATPG is repeated with some inputs forced to be X. To this end, 1 % of inputs are randomly selected and assigned to X through an ATPG constraint.

6.2. Detection Probability Computation

In this chapter, an approach for the accurate computation of the detection probability of a possibly detected fault is proposed. For each possibly detected fault there are three possibilities:

1. The fault is not detected by any of the test patterns, no matter how the X values are assigned.

2. The fault has a certain probability of being detected by at least some of the test patterns. These faults are considered to be potentially detected.
3. The fault is definitely detected by at least one test pattern, no matter how the X values are assigned.

Faults that fall into the first and last category are not actually possibly detected. However, as the experiments will show, a lot of faults that are considered to be possibly detected by a commercial ATPG can actually be placed into one of these two categories.

The difference between a *possibly* detected fault and a *potentially* detected fault lies within the detection probability. A possibly detected fault is classified by an ATPG tool which cannot prove that the fault is always detected. Thus, the classification as possibly detected does not indicate the detection probability. For a potentially detected fault, the proposed characterization approach was able to prove that the fault has a detection probability greater than zero and smaller than 100 percent.

The presented approach not only accurately computes the probability that a possibly detected fault is actually detected. It can also prove that a fault is definitely detected by a pattern or that it is definitely never detected by all patterns.

Figure 6.2 gives an overview of the overall structure of the characterization algorithm.

After a fault is chosen, all of the test patterns that possibly detect this fault are collected. Next, the detection probability for each of these patterns is computed. This is the core step (highlighted in orange) of the presented approach. If a pattern has a 100 % probability of detecting the current fault, it is definitely detected and the remaining patterns are not considered any further. Otherwise, the detection probability with the current pattern is stored and the next test pattern is evaluated. Once every pattern was considered, the overall detection probability of the fault is computed.

The computation of the detection probability of a fault for a certain test pattern consists of three different steps, each of which is further discussed in the next sections. The input of the algorithm consists of a fault and a test pattern which possibly detects the fault. First, a Boolean formula that encodes the circuit and fault detection properties is generated. Next, the variables of the Boolean formula that correspond to the circuit's inputs are restricted in accordance with the test pattern. Finally, the detection probability for the considered combination of fault and test pattern is computed.

Figure 6.3 summarizes the flow in the computation of the overall detection probability of a fault. The numbers indicate the order of the computations.

6.2.1. Generating the Boolean Formula

The computation of the detection probability of a possibly detected fault is encoded into a Boolean formula Φ as a #SAT problem. A satisfying assignment to Φ must correspond to an assignment of the X-inputs that makes the fault visible at the output. Thus, the general formula layout is similar to that utilized in SAT-based ATPG (see Section 3.1 in Chapter 3 for a more detailed discussion of the formula generation).

The final Boolean formula consists of the cells in the fault justification and fault propagation cone as well as the support cone. Again, the cells in the propagation cone are

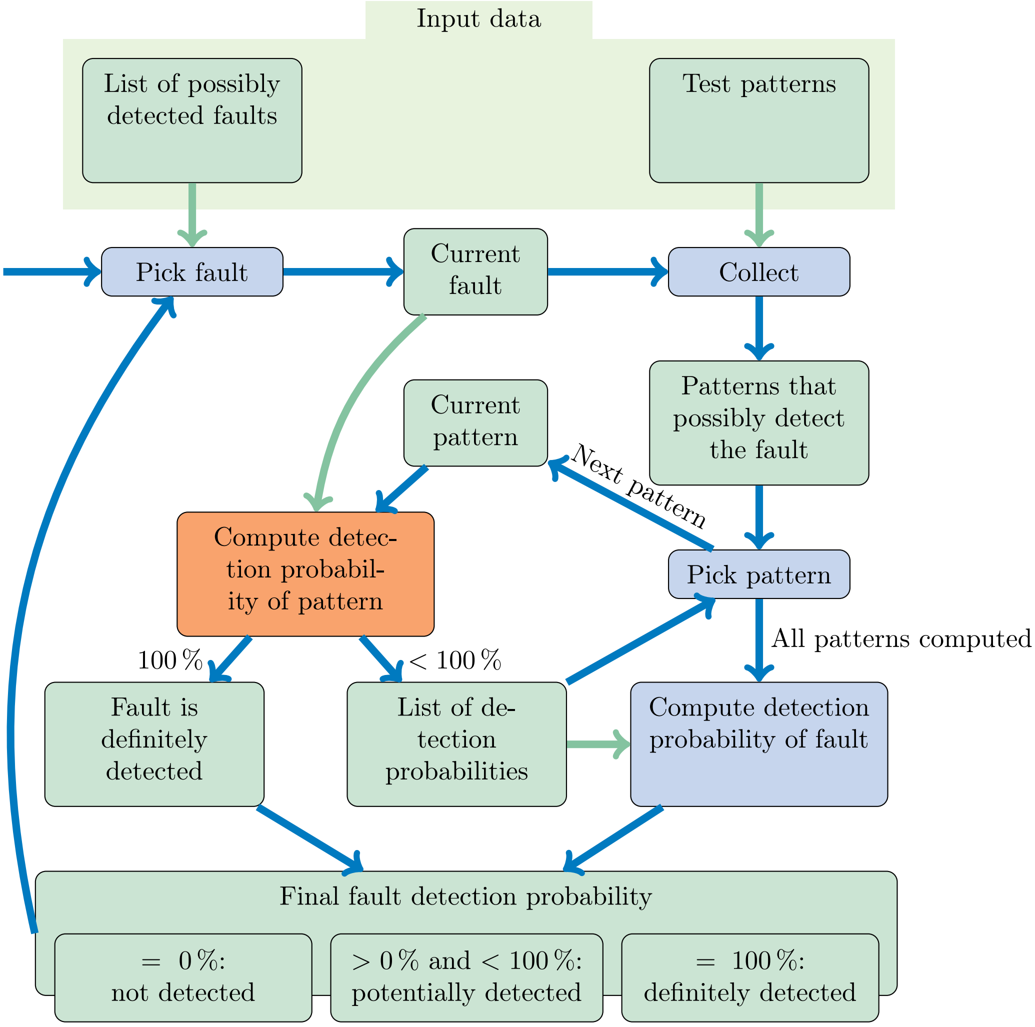

Input data
List of possibly detected faults
Test patterns
Pick fault
Current fault
Collect
Patterns that possibly detect the fault
Current pattern
Next pattern
Compute detection probability of pattern
Pick pattern
All patterns computed
100 %
< 100 %
Fault is definitely detected
List of detection probabilities
Compute detection probability of fault
Final fault detection probability
= 0 %: not detected
> 0 % and < 100 %: potentially detected
= 100 %: definitely detected

Figure 6.2.: Overview of the algorithm to characterize possibly detected faults. The blue arrows show the program flow, green arrows indicate the flow of information.

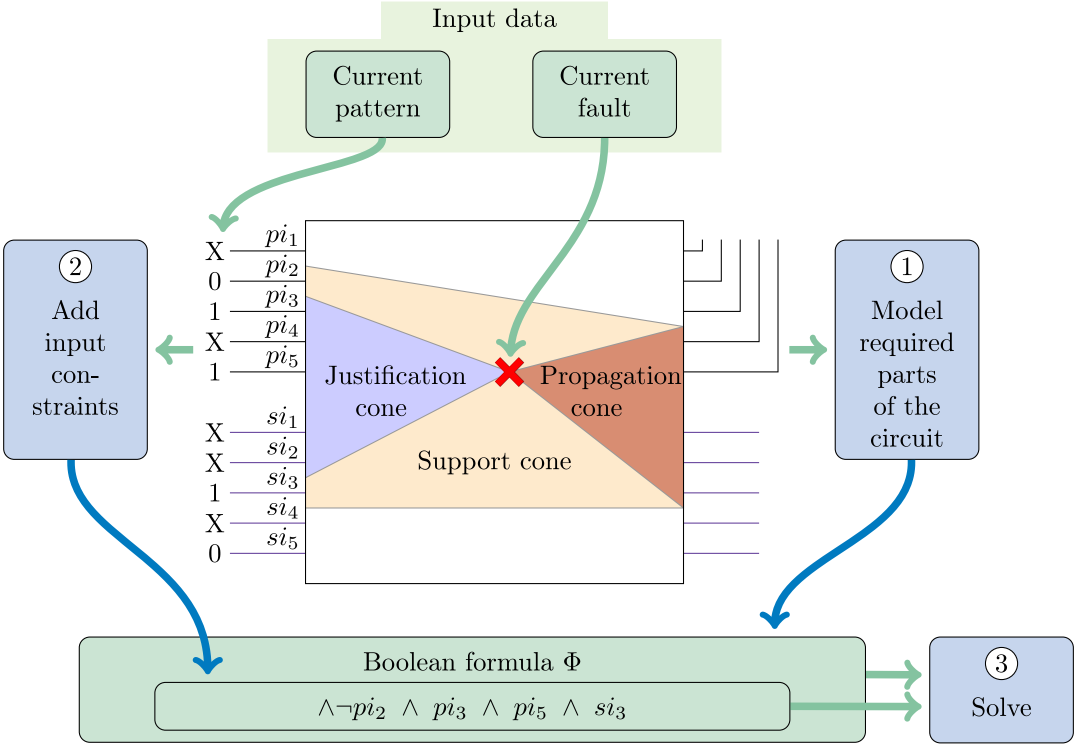


Figure 6.3.: Computing the detection probability of a fault for a given test pattern.

transformed into a Boolean formula twice, once to model the fault-free (good) circuit and once to model the signal values in the fault-affected (bad) circuit. Finally, corresponding outputs of the good and the bad circuit are connected to XOR-cells which encode the checking for a difference at the output. By forcing at least one of the variables that correspond to the outputs of these XOR-cells to be *true* through a single additional clause, it is ensured that the formula is only satisfiable if the fault effect is propagated to at least one output.

In the previously discussed optimized stuck-at ATPG, the Boolean formula is constructed incrementally. The incremental approach allows the solver to quickly find a test pattern without having to build up the entire formula. When characterizing a possibly detected fault, every single input assignment has to be considered. Therefore, an incremental formula construction is not useful.

6.2.2. Restricting the Inputs

The previously generated Boolean formula Φ encodes the general test pattern generation problem for the considered fault. Solving Φ with a SAT solver gives a test pattern that activates the fault and propagates the fault effect to at least one output. Solving Φ with a #SAT solver gives the total number of different test patterns that activate the fault and propagate its effect to an output.

While both of these results are interesting and relevant in their own right, the goal of the proposed algorithm is the characterization of a previously generated test pattern with regard to the detection probability. The test pattern contains the values of some of the circuit inputs. These inputs have to be specified in the Boolean formula since they cannot be assigned freely. Therefore, the variables that correspond to inputs that are assigned a non-X value in the test pattern are forced to the corresponding value through a unit clause. Variables that correspond to inputs with an X value are not restricted in the formula and the #SAT solver can freely chose an assignment for these variables. The assignments are only added for the inputs that are in the justification or support cone of the fault. In the example in Figure 6.3 the assignment of input si_5 to '0' is not added because the input is in neither cone.

6.2.3. Computing the Detection Probability

Solving the previously generated and restricted Boolean formula Φ with a #SAT solver gives the total number of different assignments to the X-inputs that activate the fault and propagate its effect to an output. A #SAT solver counts all different possible variable assignments that satisfy the considered formula. However, in the fault characterization formula Φ the values of the variables that were introduced during the Tseitin transformation are implied by assigning the inputs. Therefore, only the variables that correspond to the circuit inputs are truly free. Thus, each possible assignment of the X-inputs is counted only once.

The detection probability of a fault for a specific test pattern is computed by dividing the model count of Φ by the overall number of different assignments of the X-inputs:

$$detectionProbability(fault, pattern) = \frac{model\ count(\Phi)}{2^{\#X\ inputs}} \tag{6.1}$$

When counting the X-inputs it must be ensured that only inputs that are actually modeled in Φ (because they are in the justification or support cone) are considered. Otherwise the computation of the detection probability would be incorrect.

6.3. Improvements

The performance of the characterization approach for possibly detected faults that was introduced in the previous section can be further improved. This is especially important since solving the #SAT problem can be very difficult and the solve time can be substantial. This section introduces four improvements that reduce the overall complexity of the problem.

6.3.1. Restrict Propagation Outputs

The computation of the propagation cone of the fault is based on structural information only: When a cell is connected to the output of another cell in the propagation cone or to the faulty cell itself, it is also considered to be in the propagation cone. Nonetheless, it might not be possible to propagate the fault effect to every output in the propagation cone. Figure 6.4 shows an example for a fault effect that cannot be propagated to all outputs in the propagation cone. When applying the test pattern "00X" to the circuit, the output o_1 will always be '0' no matter the fault effect. Thus, this output can be ignored.

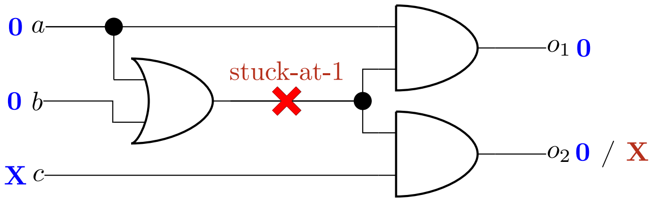


Figure 6.4.: The output o_1 is in the propagation cone of the fault, but the fault effect can never be detected there.

To reduce the search space that has to be covered by the #SAT solver, the first improvement identifies all of the outputs to which the fault effect can never be propagated, no matter how the X-inputs are assigned. To this end, the Boolean formula Φ that was created in the previous step is modified by requiring that the fault is propagated to a specific output o. The formula is then solved with a SAT solver. If the SAT solver shows that the formula is unsatisfiable, it is proven that the fault effect cannot be observed at o. This process is repeated for every circuit output that is in the propagation cone.

When computing the propagation cone for the Boolean formula for the fault characterization by the #SAT solver, outputs where the fault definitely cannot be observed are ignored. This way, the #SAT solver will not attempt to propagate the difference between the fault-free and faulty circuit to this output. Furthermore, by considering fewer outputs in the propagation cone the size of the support cone might also be reduced. Thus, the overall size of the formula might be much smaller.

If the discussed approach shows that the fault effect cannot be propagated to any output, it is proven that the fault is not detected by the current test pattern at all – without ever calling the #SAT solver.

From a cost perspective, identifying outputs to which the fault effect definitely cannot be propagated requires one SAT solver call for each output in the propagation cone. However, these calls are relatively cheap since the fault has to be propagated to a specific output.

6.3.2. Fixed X-Inputs

Not only might it not be possible to propagate a fault effect to one of the outputs in the propagation cone, it might also not be possible to freely assign all of the inputs with an X value in the test pattern. Instead, to initialize the fault or to propagate its effect, some of these inputs might require a fixed assignment to either '1' or '0'. As an example, consider the stuck-at fault in Figure 6.4 again. This fault can only be detected when assigning the input c to '1'.

The identification of such fixed X-inputs is, again, performed by utilizing a SAT solver. For each X-input the formula Φ is extended, once with the condition that this input is assigned to '0' and once with the condition that it is assigned to '1'. The resulting formulas are solved with a SAT solver. If the formula is unsatisfied when the input is assigned to '0', the X-input always has to be assigned to '1' and vice-versa. When a fixed X value is found, the required assignment is added to Φ as a unit clause.

In addition, if Φ is unsatisfied when the input is assigned to '0' as well as when it is assigned to ‘1’, it is proven that the fault cannot be detected by the current test pattern at all, since there is no way to assign the current X-input.

From a cost perspective, identifying fixed X-inputs requires two SAT solver calls for each X-input in the justification and support cone.

6.3.3. Always Satisfied Formulas

In addition to reducing the number of outputs to which a fault can propagate and to forcing some X-inputs to a fixed value, the two previously discussed improvements are also capable of detecting that some faults are never detected by the test pattern. The third improvement aims at identifying the opposite: Faults that are always detected by the test pattern, no matter how the X-inputs are assigned [99].

When the fault is always detected this also means that there exists no assignment of the X values that does not satisfy the formula. No matter how the X-inputs are assigned, the formula will always be satisfied. This interpretation can easily be encoded as a SAT problem by slightly modifying Φ. Instead of requiring that the fault effect is propagated to at least one output, it is required that the fault does not propagate to any output at all. If the resulting formula is unsatisfiable, it is proven that no assignment to the X-inputs exists that does not detect the fault. Thus, every single X value assignment successfully tests the considered fault and the fault is always detected.

In combination with the identification of fixed X-inputs the detection probability of an always detected fault for a given pattern is not necessarily 100 %. Consider a case where the previous optimization found that one of the X-inputs has to be fixed to ‘1’. After adding this fixed input requirement to Φ, every assignment to the remaining X-inputs successfully tests the fault. In this case, the detection probability for the fault would be 50 %.

From a cost perspective, testing if the fault is detected no matter how the X-inputs are assigned, requires only a single SAT solver call.

6.3.4. Caching the Detection Probability

To compute the detection probability of a possibly detected fault, only the inputs in the justification and support cones need to be considered. All other primary and secondary circuit inputs are completely irrelevant for this particular fault. This restriction to a subset of the circuit inputs also reduces the number of relevant input assignments in the test patterns. Thus, it is possible that two different test patterns create the exact same input assignment at the considered circuit inputs. Clearly, the detection probability of the possibly detected faults for these two test patterns will be identical. Therefore, a cache that stores the results of all previously considered test patterns for a fault is added to the presented characterization approach.

In addition to storing the detection probability, the cache is also used to remember especially difficult to compute problems that result in a timeout. If a test pattern has the exact same input assignment at the relevant inputs of the circuit, the algorithm does not waste any valuable computation time but instead directly marks the current pattern as having a time-out as well.

6.4. Evaluation

This chapter introduced a novel approach to accurately characterizing possibly detected faults. In this section the presented approach is evaluated. To this end, a commercial ATPG tool is used to generate stuck-at test patterns for a large collection of different circuits. All of the faults that are only possibly detected by the commercial ATPG tool are then analyzed with the proposed algorithm.

All of the tables referenced in the evaluation are printed at the end of the chapter.

6.4.1. Characterizing Possibly Detected Faults

For the first set of experiments the commercial tool is used with its default settings to generate test patterns. Table 6.2 shows the number of generated test patterns as well as the number of possibly detected faults for each of the circuits. Furthermore, it gives the average number of test patterns that detect each of the possibly detected faults.

For some of the circuits the ATPG did not classify any faults as possibly detected. In case of the AES circuits this can be explained by the high combinational complexity which makes an X value more likely to propagate through the entire circuit. Thus, an X value is likely to make the fault completely untestable from the ATPG perspective, because both the good and bad output value is X. Circuits without any possibly detected faults are skipped for the evaluation.

Most possibly detected faults are possibly detected by more than one test pattern, especially on the larger circuits. This makes the analysis of the detection probability more expensive. For example for the circuit $p388k$ which has 1 848 possibly detected faults, each of which is possibly detected by 12.66 patterns on average, a total of $1\,848 \cdot 12.66 \approx 23\,395$ different combinations of faults and test patterns have to be analyzed.

All of the possibly detected faults are characterized with the proposed algorithm with the solver `countAntom` in single-threaded mode. As shown in the previous chapter, `dCountAntom` can be used to increase the scalability by utilizing cluster scale computing. The possibility of using many CPU cores in parallel to tackle the more difficult formulas is discussed in Section 6.4.5. The soft timeout of `countAntom` is set to 5 minutes, the hard timeout to 30 minutes.

Table 6.3 summarizes and Figure 6.5 visualizes the results of the characterization of the possibly detected faults into one of the three previously introduced categories: definitely detected, potentially detected and definitely not detected. Faults that could not be characterized due to timeouts are added to the fourth category *unknown*.

Of the faults that are considered to be possibly detected by a commercial ATPG, a large amount is actually definitely detected by at least one test pattern. Of the remaining faults the majority is potentially detected. For most circuits only a small share of faults is definitely not detected. The exact composition of the different faults is circuit-dependent. For some circuits almost all faults are definitely detected or definitely not detected (e.g.,

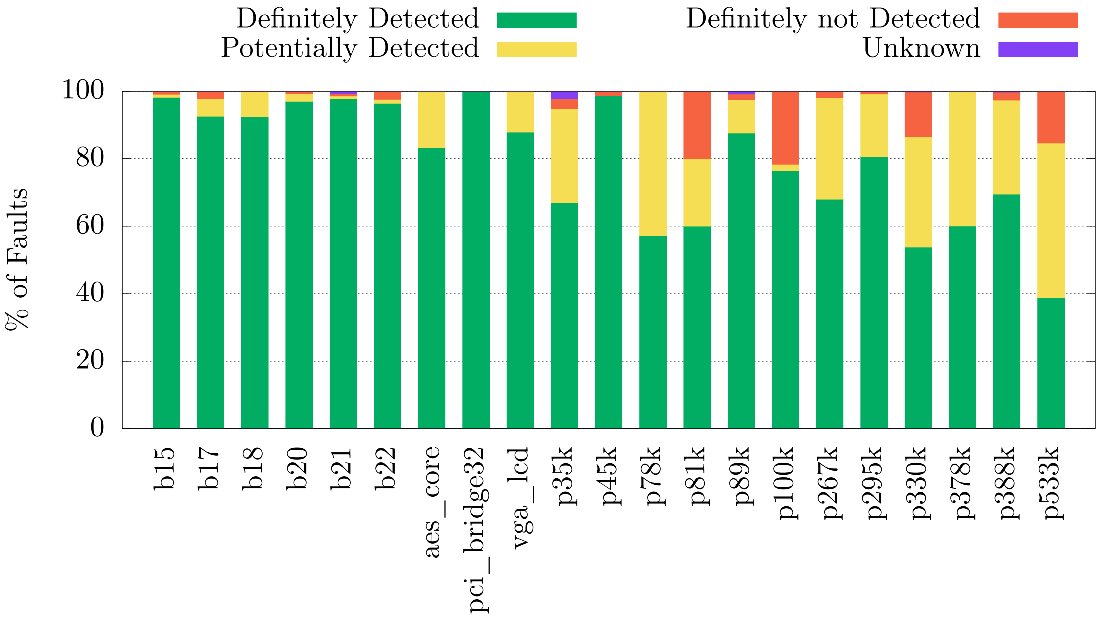


Figure 6.5.: **Characterization** of the possibly detected faults in each circuit.

*b*20, *p*100*k*, ...), whereas for other circuits a large amount of faults is potentially detected (e.g., *p*78*k*, *p*533*k*, ...).

6.4.2. Detection Probability

Many faults are possibly detected by more than one test pattern. The overall detection probability of a fault is composed of the individual detection probabilities of each of these possibly detecting test patterns and is computed in two different manners in this work.

The pessimistic detection probability estimation uses the highest detection probability of a test pattern as the overall detection probability of the fault.

The optimistic detection probability estimation, on the other hand, assumes that all of the detection probabilities are stochastically independent and computes the joined probability that the fault is detected based on that assumption.

As an example, assume that a fault is potentially detected by two different test patterns with a probability of 50 % each. The pessimistic estimate for the overall detection probability would be computed as 50 % while the optimistic estimate would be 75 %. The real detection probability is, most likely, somewhere between those two values.

Figure 6.6 gives an overview of the average overall detection probability for each circuit for both the pessimistic and optimistic estimation.

For some faults not all test patterns could be evaluated due to timeouts. If at least one of the test patterns that possibly detects the fault can be characterized without a timeout, the coverage is counted towards the overall average coverage. A more detailed analysis of the timeouts is performed in Section 6.4.4.

The average detection probability of the possibly detected faults is highly circuit-dependent. Furthermore, since many faults are possibly detected by multiple test patterns with different detection probabilities, the pessimistic and optimistic estimates for the

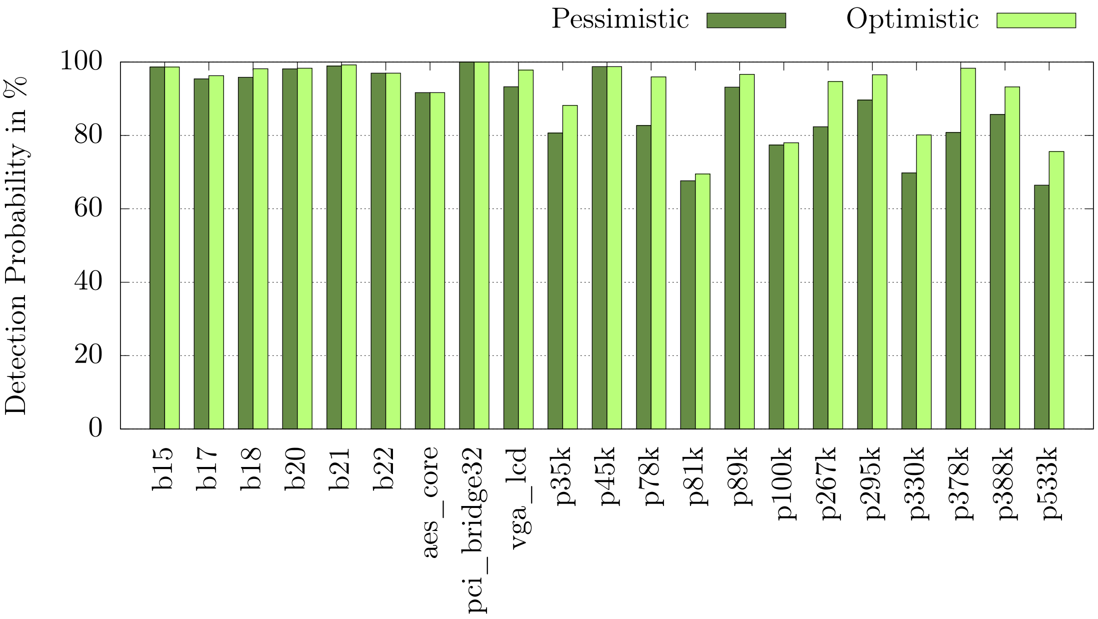


Figure 6.6.: Average overall **detection probability** of **all possibly** detected faults.

detection probabilities are different as well. The pessimistic detection probability estimate ranges from a minimum of 66.4 % up to 100 % with an average of 87.8 %. The optimistic estimate gives an average of 92.5 %.

For the computation of the average detection probability shown in Figure 6.6, all of the faults that were successfully characterized are considered. These include the faults that are definitely detected by at least one test pattern and have a 100 % detection probability. Figure 6.7 shows the average detection probability for only the potentially detected faults. Clearly, when not considering the large number of definitely detected faults, the average detection probability is much lower. What is more, the differences between the pessimistic and optimistic detection estimates become more pronounced.

The large differences between the circuits make it impossible to predict the overall detection probability without any additional knowledge. Therefore, simply counting a fraction of the possibly detected faults towards the overall fault coverage – which is the default setting in some commercial ATPGs – can be dangerous since it might overestimate the real coverage of these faults. On the other hand, considering possibly detected faults as definitely not covered needlessly underestimates the real fault coverage.

6.4.3. Optimizations

The proposed #SAT-based approach is able to accurately characterize faults that are considered as possibly detected by a commercial ATPG tool. In addition, this work also presented multiple SAT-based improvements. This section analyzes the gains provided by the different improvements.

The number of outputs that need to be considered for the fault observation and the number of X-inputs that can be chosen freely by the #SAT solver are both reduced.

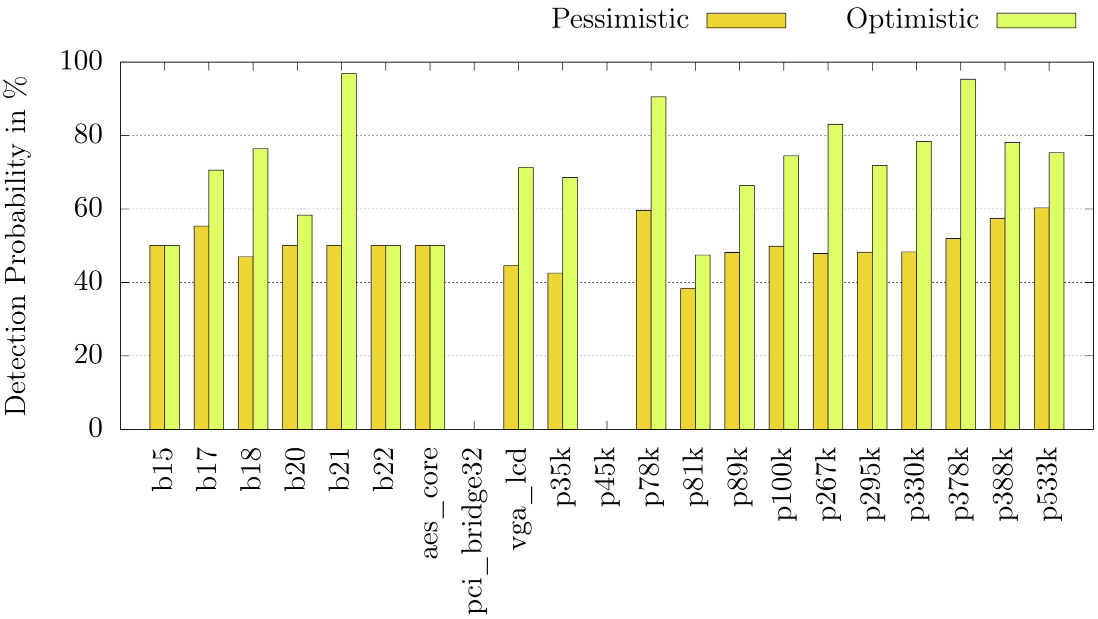


Figure 6.7.: Average **detection probability** of the **potentially** detected faults.

Figure 6.8 shows the achieved reduction in the number of modeled outputs and free X-inputs.

Clearly, the proposed improvements greatly decrease the complexity of the formula that is handed over to the #SAT solver. On average across all circuits, the number of outputs that need to be considered is reduced by about 24.9 % and the number of free X-inputs by about 8.0 %. The gains of these two optimizations highly depends on the circuit.

The previous optimizations, furthermore, sometimes quickly characterize a fault as definitely not detected. In combination with the quick discovery of always satisfied formulas these improvements greatly reduce the number of #SAT solver calls. In addition, some computations can be skipped entirely because the exact same assignment to the relevant inputs has been characterized before and the result is cached.

Figure 6.9 shows that the majority of formulas can be solved without ever calling the #SAT solver because one of the previously introduced improvements already computed the outcome. Only the formulas in the yellow "remaining" block are passed to the #SAT solver. This greatly increases the overall solving speed as the #SAT solver is only used for non-trivial problems which reduces the overhead.

A large share of formulas – about 78.0 % on average – is always satisfied. However, this does not mean that the corresponding fault has a detection probability of 100 % since the number of free X-inputs might be restricted through the presented optimizations. The caching mechanism is only rarely used. The largest utilization is observed on the circuit $p533k$ where about 39.3 % of the computations are skipped.

6.4.4. Timeouts

Although the proposed possibly detected fault characterization approach contains multiple optimizations and is based on an efficient #SAT solver, there are still some formulas that

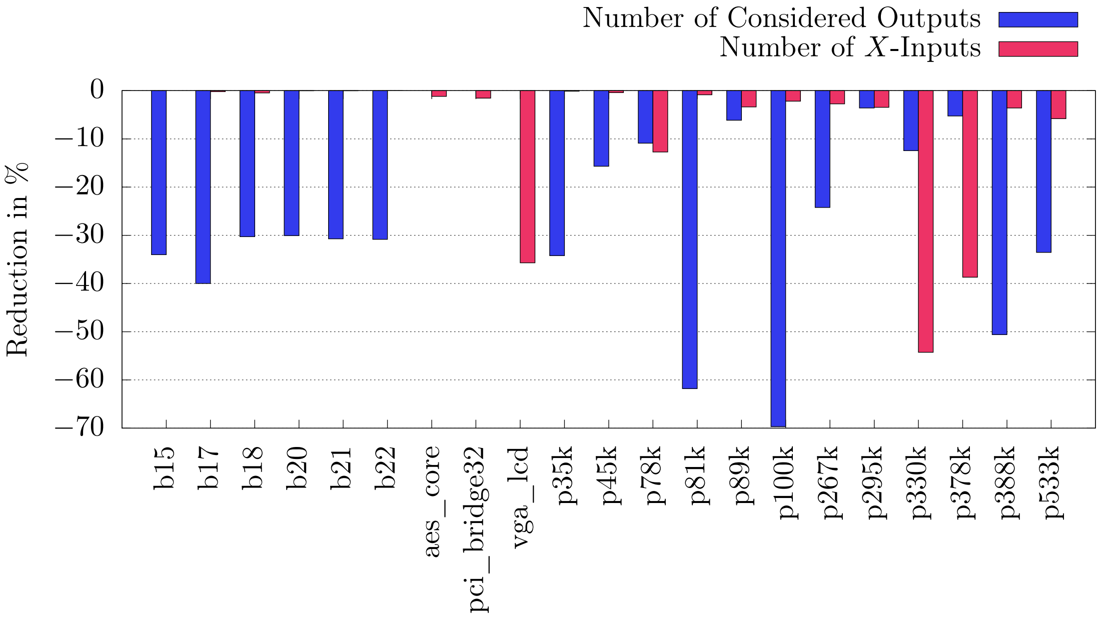


Figure 6.8.: Average reduction of the number of outputs that can show a difference and of the number of X-inputs that can be chosen freely.

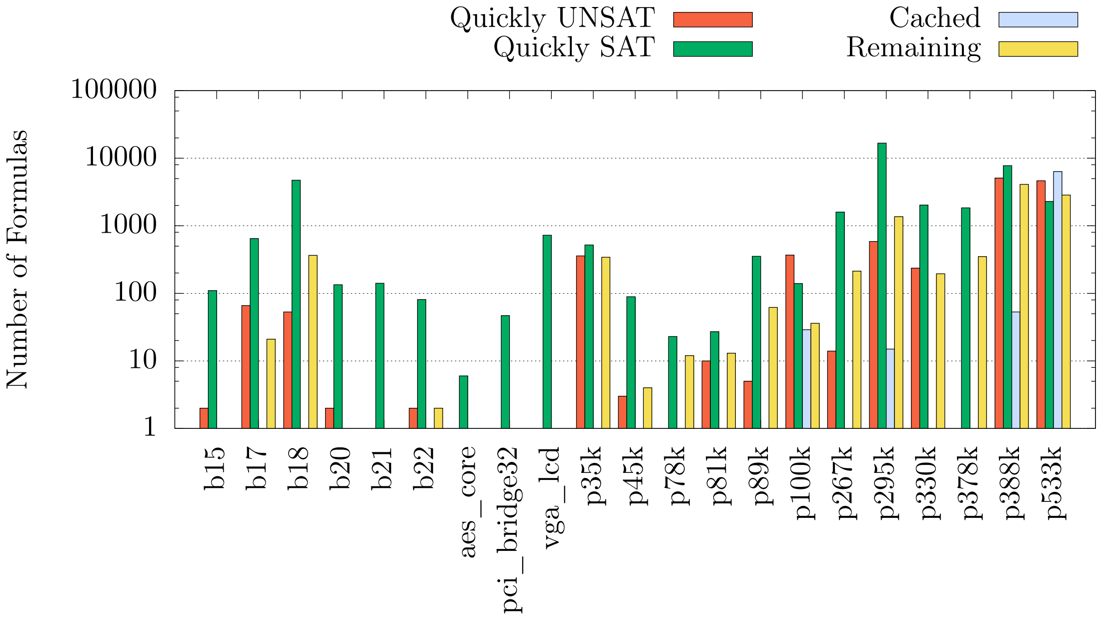


Figure 6.9.: Some **formulas can be quickly solved** through one of the optimizations. The remaining formulas are solved with the #SAT solver.

cannot be solved sufficiently quickly and a timeout occurs. For some faults the timeout is irrelevant because they are definitely detected by another pattern. The detection probability (100 %) of these faults can be considered for the overall coverage without hesitation.

For other faults the detection probability for at least some of the test patterns can be computed without a timeout. By at least considering the detection probability of these patterns, the overall average detection probability more accurately reflects all of the faults, including the difficult ones. Nonetheless, the average might be slightly worse because one of the patterns with a timeout might have a higher detection probability.

Figure 6.10 shows the number of faults that cannot be characterized at all because of timeouts as well as the number of faults for which at least some timeouts occur for each circuit.

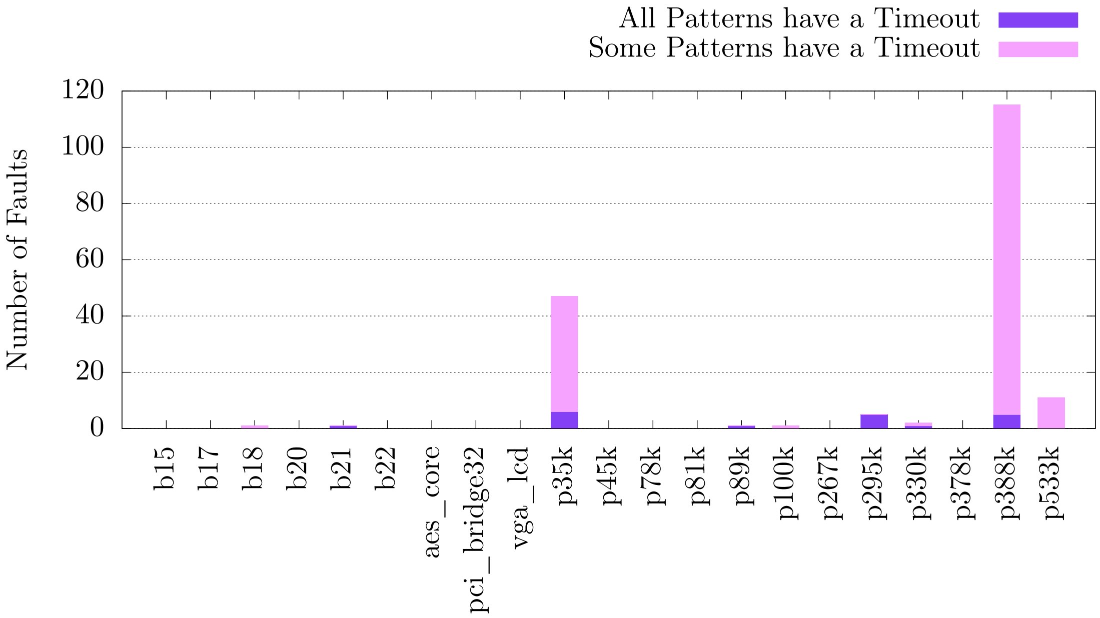


Figure 6.10.: Number of **faults with a timeout** for every test pattern or for some of the test patterns that possibly detect the fault.

The number of timeouts across the different circuits is very low. Furthermore, at most six faults (on the circuit *p*35*k*) cannot be characterized at all due to timeouts. Usually, when timeouts occur there is at least one test pattern for which the fault can be characterized.

Unlike classic #SAT solvers, `countAntom` contains a much more advanced timeout system with soft and hard timeouts. The computation can be aborted after 5 minutes already, if it seems unlikely that the formula will be solved within 30 minutes. Otherwise, the solve process can continue for up to 30 minutes.

Figure 6.11 shows the number of formulas that is solved after the soft timeout was exceeded because the predicted total solve time is below the hard timeout. In addition, the Figure also shows the number of formulas that were aborted early after the soft timeout because it was unlikely that they will be solvable before the hard timeout.

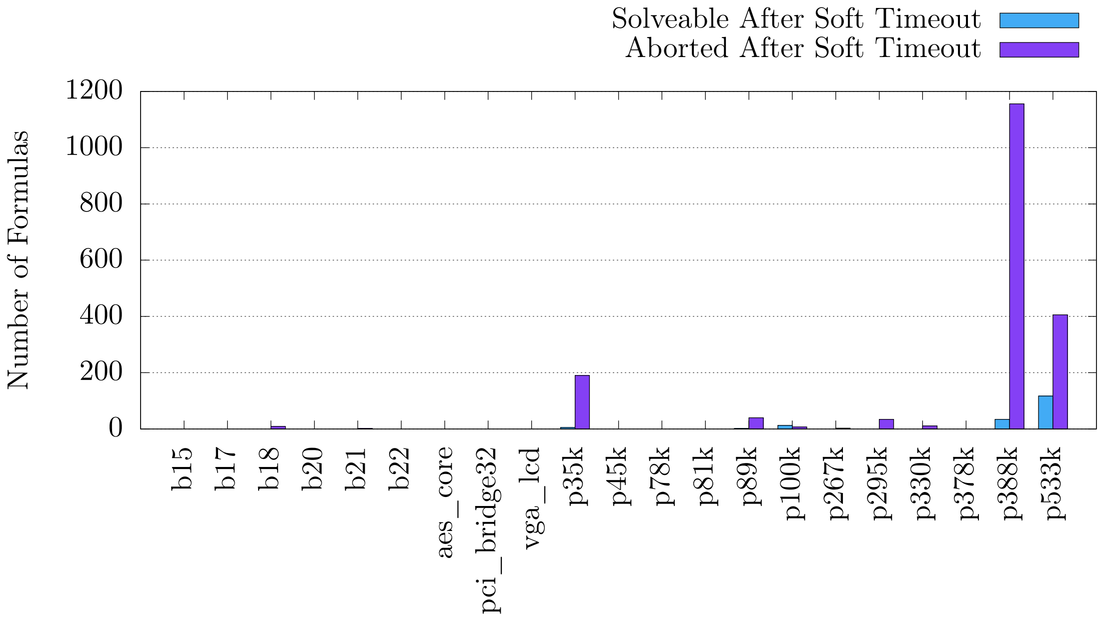


Figure 6.11.: Number of formulas that are solved after the soft timeout and number of formulas that are aborted early, after the soft timeout, already.

The number of timeouts is higher than the number of faults that could not be characterized due to timeouts. This is because some of the faults for which a timeout occurred are definitely detected by another test pattern and the timeout is irrelevant.

To understand the benefits of the soft timeout mechanism consider the circuit $p533k$. In total 117 formulas are solvable after the soft timeout of 5 minutes but before the hard timeout of 30 minutes. At the same time, the calculation of 406 formulas is aborted after the soft timeout of 5 minutes because it is unlikely that they will be solved within 30 minutes. To solve the 117 formulas that could be solved within 30 minutes in a classic #SAT solver with only a hard timeout this timeout would have to be set to 30 minutes. Thus, every formula would be analyzed for up to an additional 25 minutes before the timeout occurs.

When assuming that none of the 406 formulas that were aborted by `countAntom` after the soft timeout are actually solvable within 30 minutes, this would have increased the solving time by $406 \cdot 25 = 10\,150$ minutes or about 169 hours. With the soft timeout mechanism unique to `countAntom` and `dCountAntom` this large increase in characterization time (without any additional gains) is avoided.

6.4.5. Solve Time

For each combination of fault and test pattern the #SAT solver is called at most once. However, the SAT solver is usually utilized to perform a large number of different computations for the optimizations. Figure 6.12 shows the average time used for the different optimizations and the accurate computation of the detection probability through the #SAT solver.

For most of the circuits the #SAT solve time dominates the overall computation time for each combination of fault and test pattern. This was to be expected, as all of the im-

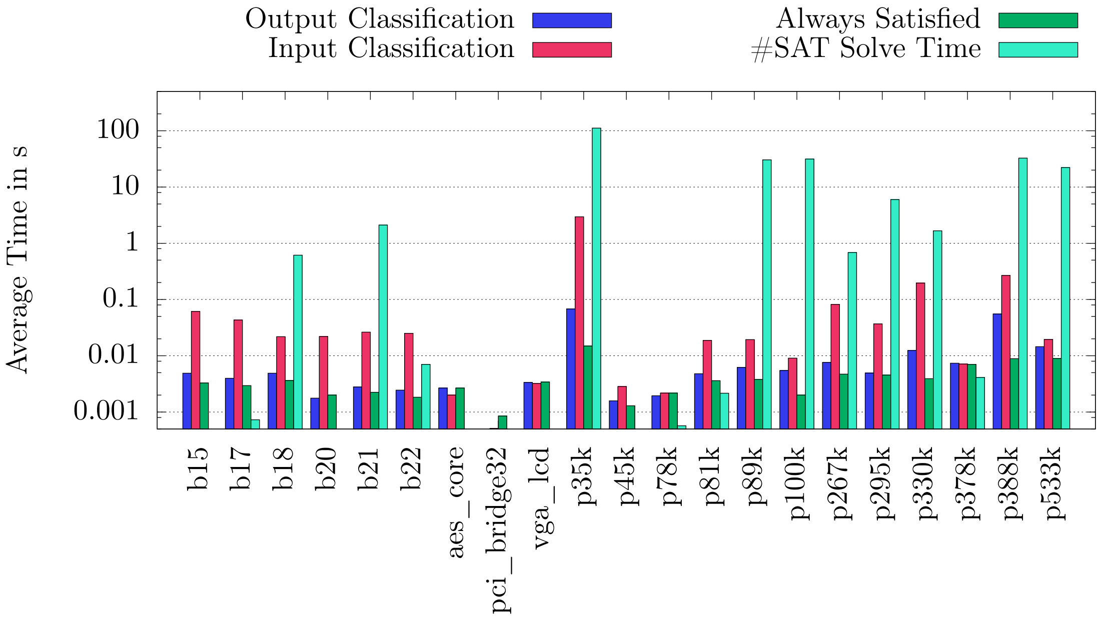


Figure 6.12.: The **average computation time** for each combination of fault and test pattern for the different improvements and the #SAT solver.

provements are encoded as simple SAT solver calls. Of the improvements, the computation of fixed X-inputs takes up the most time. This was also expected since this optimization utilizes the SAT solver extensively with many calls (two for each X-input).

Because the #SAT solve time is generally dominant, it is the best starting point for optimizations. In the previous chapter the distributed parallel #SAT solver `dCountAntom` was introduced and evaluated. By utilizing cluster-scale computing, the presented characterization approach can be scaled to handle the more complex formulas much quicker. Table 6.1 summarizes the solve times for some of the hard to solve formulas when the #SAT solver is used in parallel.

Table 6.1.: Comparison of the #SAT solver solve time with 1 and 256 cores.

	Solve time (s)	
Formula	`countAntom` 1 core	`dCountAntom` 256 cores
$p35k$, fault 69	1 135.4	10.7
$p100k$, fault 82	1 970.6	16.3
$p388k$, fault 71	843.8	11.0
$p533k$, fault 314	1 701.6	18.3

With 256 CPU cores, all of the analyzed, difficult to solve formulas can be solved within 20 s. This clearly shows that the bottleneck of the #SAT solver can be skirted by utilizing more resources. This way, not only can the overall computation time be drastically reduced, it is also possible to reduce the number of timeouts.

6.4.6. Inputs that are Always X

So far, possibly detected faults arose because of decisions by the commercial ATPG tool – there was no apparent reason to assign some of the inputs of a test pattern to X.

However, as discussed at the beginning of the chapter, in a real circuit it might occur that some of the circuit's inputs cannot be set to a fixed value at all. These inputs could, for example, be connected to a non-scan flip-flop or to another device that cannot be controlled while applying the test pattern. In these cases some of the inputs are always unknown, X. To simulate a circuit with some of the aforementioned test restrictions, 1 % of the primary and secondary inputs are chosen at random for each circuit. The test pattern generation with the commercial ATPG tool is then repeated with the restriction that these inputs are always assigned to X.

Table 6.4 gives an overview of the number of test patterns and possibly detected faults as well as the average number of patterns that possibly detect each possibly detected fault.

The characterization of the possibly detected faults is visualized in Figure 6.13. Compared to the unrestricted ATPG the number of possibly detected faults has grown for almost all circuits. Furthermore, the number of test patterns that possibly detect each of these faults is also increased. To compute the detection probability for such a large number of combinations of faults and test patterns, the soft timeout is reduced to 30 seconds with a hard timeout of 3 minutes.

Even with the reduced computation time, the number of faults that could not be characterized due to timeouts is generally low with, on average, only 3.4 %. The circuits $p35k$ and $p89k$ are the exception from the previous observation. Here, timeouts actually occur for a larger share of faults. This might be because of the random selection of highly relevant inputs to be assigned to X which complicates the computation.

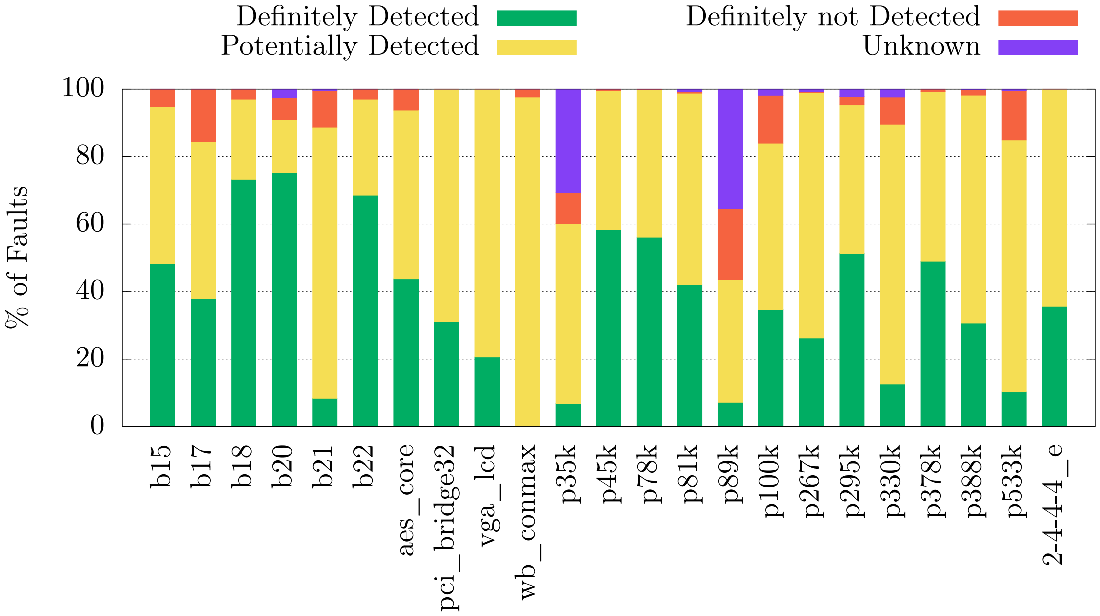


Figure 6.13.: **Characterization of the possibly detected** faults when 1 % of circuit inputs are forced to be X.

Figure 6.14 shows the average detection probability for each circuit. In comparison to the detection probability without any input constraints during the test pattern generation, the detection probability is more variable and circuit-dependent. The detection probability is, additionally, much lower with 60.0 % and 76.4 % on average for the pessimistic and optimistic estimate.

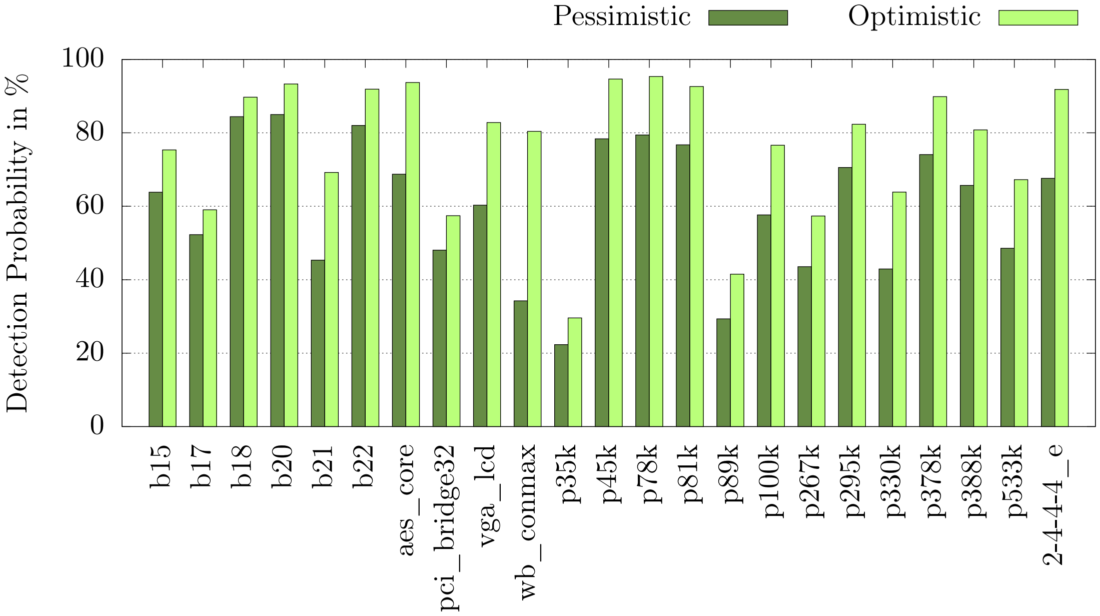


Figure 6.14.: Average overall **detection probability of the possibly detected faults** when 1 % of circuit inputs are forced to be X.

Overall, these results show that forcing some inputs to be X during the pattern generation increases the uncertainty. For some circuits the detection probability estimate is well below 50 %, whereas for others it is above 90 %. This makes an accurate characterization of the possibly detected faults even more relevant.

6.5. Summary

This chapter introduced and evaluated a novel method to characterize possibly detected faults that allows for an accurate computation of the detection probability of the analyzed faults.

The key points of the chapter are as follows:

- Faults that a commercial ATPG algorithm can only classify as possibly detected can be accurately characterized by the proposed #SAT-based approach for the first time. It allows for the computation of the detection probability for each of the possibly detected faults.
- The average detection probability of the possibly detected faults is highly circuit-dependent and cannot be estimated without additional knowledge.

- The presented approach scales well even to large industrial circuit.
- The developed improvements which utilize a SAT solver simplify the problem and, furthermore, quickly identify some always or never detected faults.

The presented approach allows for an accurate reasoning about the detection probability of a fault. In the future, such an accurate analysis could be used to further increase the fault coverage by creating additional test patterns or placing test points for faults with a low detection probability. Furthermore, the algorithm itself could be improved by including an accurate fault simulator with support for X values to perform a pre-characterization. In addition, applying the approach to different fault models might yield further insights into the importance of accurately considering possibly detected faults. Since every possibly detected fault has to be analyzed separately, the characterization could also be performed for multiple faults in parallel.

Overall, this chapter highlighted that applying a #SAT solver to problems in the realm of circuit test can provide valuable information. This initial example will hopefully spawn additional interesting applications that step beyond the binary world of SAT and towards probabilities.

Table 6.2.: Results of the test pattern generation by a commercial ATPG without any input restrictions.

	Circuit	# Test Patterns	# PD Faults	#Patterns per PD Fault
ITC'99	b15	468	111	2.11
	b17	745	566	2.97
	b18	857	3 234	4.09
	b20	618	134	2.14
	b21	644	139	2.05
	b22	525	83	2.48
IWLS	aes_core	386	6	1.67
	pci_bridge32	297	40	3.92
	vga_lcd	1 561	66	13.42
	wb_conmax	224	0	-
NXP	p35k	1 100	273	8.37
	p45k	2 116	80	12.53
	p78k	139	7	6.43
	p81k	408	20	4.15
	p89k	649	121	5.24
	p100k	2 057	157	9.84
	p267k	926	210	13.54
	p295k	1 901	4 067	8.55
	p330k	1 874	431	7.86
	p378k	258	268	9.35
	p388k	941	1848	12.66
	p533k	936	508	41.11
AES	2-2-2-8_d	706	0	-
	2-2-2-8_e	678	0	-
	2-4-4-4_d	69	0	-
	2-4-4-4_e	73	0	-
	10-2-2-4_d	69	0	-
	10-2-2-4_e	74	0	-
	10-2-4-4_d	73	0	-
	10-2-4-4_e	73	0	-

Table 6.3.: Characterization of the possibly detected faults and average detection probability with the pessimistic and optimistic estimation.

	Circuit	# Definitely Detected	# Potentially Detected	# Definitely Not Detected	# Unknown	Avg. Det. Probability (%) Pessimistic	Avg. Det. Probability (%) Optimistic
ITC'99	b15	109	1	1	0	98.65	98.65
	b17	524	29	13	0	95.42	96.30
	b18	2 988	239	7	0	95.86	98.19
	b20	130	3	1	0	98.13	98.32
	b21	136	1	1	1	98.91	99.25
	b22	80	1	2	0	96.99	96.99
IWLS	aes_core	5	1	0	0	91.67	91.67
	pci_bridge32	40	0	0	0	100.00	100.00
	vga_lcd	58	8	0	0	93.28	97.82
NXP	p35k	183	76	8	6	80.65	88.18
	p45k	79	0	1	0	98.75	98.75
	p78k	4	3	0	0	82.70	95.95
	p81k	12	4	4	0	67.66	69.49
	p89k	106	12	2	1	93.15	96.65
	p100k	120	3	34	0	77.39	78.02
	p267k	138	61	4	0	82.37	94.69
	p295k	3 265	756	28	5	89.65	96.54
	p330k	232	141	57	1	69.80	80.17
	p378k	161	107	0	0	80.81	98.33
	p388k	1 284	515	44	5	85.73	93.24
	p533k	197	233	78	0	66.44	75.61

Table 6.4.: Results of the test pattern generation by a commercial ATPG when 1 % of primary and secondary circuit inputs are always assigned to X.

	Circuit	# Test Patterns	# PD Faults	#Patterns per PD fault
ITC'99	b15	442	58	8.24
	b17	631	116	4.97
	b18	1090	1 827	11.31
	b20	636	77	25.52
	b21	638	2 555	11.88
	b22	569	232	13.56
IWLS	aes_core	375	16	17.44
	pci_bridge32	303	87	8.67
	vga_lcd	1 544	223	4.60
	wb_conmax	255	43	24.77
NXP	p35k	1 043	1 387	10.51
	p45k	2 094	226	15.69
	p78k	162	460	13.45
	p81k	509	2939	26.21
	p89k	639	643	8.75
	p100k	2 036	542	14.23
	p267k	924	1 270	16.81
	p295k	940	3 257	16.02
	p330k	1 526	2 077	11.16
	p378k	300	3 095	9.07
	p388k	912	5 823	16.09
	p533k	987	9 363	30.72
AES	2-2-2-8_d	0	0	-
	2-2-2-8_e	0	0	-
	2-4-4-4_d	0	0	-
	2-4-4-4_e	118	263	9.33
	10-2-2-4_d	0	0	-
	10-2-2-4_e	0	0	-
	10-2-4-4_d	0	0	-
	10-2-4-4_e	0	0	-

7. Conclusion

Testing an integrated circuit for the existence of manufacturing defects is of the utmost importance since defects are the norm and not the exception. With an ever increasing complexity and smaller and smaller feature sizes the potential for such defects is increasing and will most likely continue to grow in the foreseeable future.

At the same time, the test complexity is increasing as well. While the stuck-at fault model was initially sufficient to detect most defective devices, nowadays more and more advanced and more realistic fault models are considered. Creating test patterns for such advanced fault models is harder as additional side constraints must be taken into account for a successful test. Dealing with such constraints in classic ATPG algorithms is challenging.

This thesis focused on the utilization of SAT-based methods in the realm of circuit test. While the general application of a SAT solver for the test pattern generation has been well established in literature, the versatility of Boolean formulas, as the underlying abstract mathematical model, allows for the efficient modeling of advanced features and conditions that are not considered in many other ATPG algorithms.

In this work, novel approaches in three different major areas were introduced:

- The concept of D-chains was extended with a good-diff D-chain and novel hybrid D-chains that combine the positive traits of the different D-chain techniques. Furthermore, the first comprehensive comparison of different D-chain techniques was performed. The experimental results showed that not every D-chain is similarly well suited for different tasks and that the new hybrid D-chain approach provides the overall best improvement in solving speed.

  The performance gains of D-chains make them important for an efficient SAT-based ATPG algorithm and should be considered no matter what fault model is utilized.

- Based on an accurate timing-aware model of the circuit, the first deterministic, glitch and charge-sharing aware TSOF ATPG was introduced. Test patterns that are produced by this SAT-based ATPG algorithm are not invalidated by glitches assuming the given timing model and consider charge-sharing conflicts. The ATPG also allows for the generation of patterns that are more robust against small variations in the circuit's timing by requiring minimum glitch and initialization length durations.

  The experimental results showed that the presented approach even scales to large circuits and circuits with a high combinational complexity. Moreover, the importance of considering glitches during the test pattern generation was highlighted by showing that over 25 % of conventionally generated test patterns might be invalidated due to glitches. Furthermore, the algorithm supports the generation of test patterns that utilize glitches to initialize faults or to mitigate charge-sharing conflicts. Thus, the resulting test patterns are able to test faults that are conventionally considered as untestable.

The results obtained for the TSOF ATPG are of high relevance since many of the recent, more advanced fault models might be susceptible to similar glitch constraints and a timing-unaware test pattern generation will often not be sufficient to guarantee a successful test.

- By stepping beyond the binary yes or no answers of a SAT solver towards reasoning about probabilities, the #SAT problem opens up a whole realm of new possibilities. In this thesis, the accurate characterization of possibly detected faults was analyzed. Being closely related to the test pattern generation, many of the general techniques that are used for SAT-based ATPG can be transferred to this field of research.

  The experimental results showed that it is indeed possible to characterize most of the faults that a commercial ATPG can only classify as possibly detected. The results also revealed that the probability of detecting a possibly detected fault is highly circuit-dependent and cannot be estimated by a simple weighting factor without any additional knowledge.

  Moreover, the massive parallelization of the #SAT solve process was successfully performed and analyzed. The results showed that, given a sufficient amount of computation power, even very difficult problems can be solved successfully. The proposed methodology scales very well even with a large amount of CPU cores. In addition, the solve progress estimation and total solve time estimate were shown to be good predictions. Based on this prediction, the soft timeout mechanism was successfully applied to the characterization of the possibly detected faults.

The experiments that were performed in this work highlight that SAT and #SAT-based approaches are very powerful and can provide relevant information that would otherwise not be available. In addition, the simplicity of adding constraints and conditions to a Boolean formula makes such approaches easily extensible and upgradeable should any of the requirements change.

This extensibility also allows for a multitude of possible future research. As the work on D-chains has shown, there are still many opportunities to greatly increase the solve speed by developing and improving supplementary data structures that guide the solve process. The TSOF ATPG showed how advanced fault models with complex requirements can be incorporated in a SAT-based ATPG flow. Here, an analysis of different fault models might yield further insights into the real requirements for a successful fault detection. Finally, the application of a #SAT solver in the realm of circuit test is only in its infancy. This thesis has shown the great potential of #SAT by extracting relevant information in an efficient manner. There might be many more areas that require an accurate computation of probabilities that could benefit from applying a #SAT solver.

Overall, this thesis clearly shows that by applying SAT-based methods to circuit test one can reason about advanced and complex effects and gain additional relevant insights.

Appendix

A. Experimental Setup

The experiments are performed on four identical compute nodes each of which is equipped with a single quad-core Intel Xeon E5-2643 CPU clocked at 3.3 GHz. To evaluate the distributed parallel #SAT solver `dCountAntom` the computational resources of the bwUniCluster are used which provides 512 homogeneous servers with two octa-cora Intel Xean E5-2670 CPUs. The servers are connected by an InfiniBand 4X FDR interconnect.

A.1. Solver Description

- `antom` [13], [14] is a SAT solver that supports an incremental solving mode and assumptions. It is developed at the University of Freiburg by Tobias Schubert and Sven Reimer.
- `countAntom` [C4] is a multi-threaded #SAT solver based on `antom`. The development of `countAntom` started in the scope of the Master thesis by the author. It is further extended by improvements discussed in this work.
- `dCountAntom` [C5] is a distributed parallel #SAT solver based on `countAntom` that is described in this work.

A.2. Circuit Information

In this thesis benchmark circuits from four different sources are used to provide a thorough and extensive evaluation of the proposed algorithms. The circuits are from the ITC'99 [100] and IWLS 2005 [101] benchmark collections, industrial designs from NXP as well as combinational small scale AES implementations [W2]. Table A.1 summarizes the most relevant circuit information.

Table A.1.: Overview of the considered benchmark circuits.

	Circuit	# Inputs	# Flip-Flops	# Gates	% Complex
ITC'99	b15	37	449	3 395	51.05
	b17	38	1 414	11 345	50.38
	b18	38	3 270	34 936	45.28
	b20	33	490	5 844	46.92
	b21	33	490	5 899	47.04
	b22	33	703	8 144	46.67
IWLS	aes_core	259	530	11 082	38.43
	pci_bridge32	161	3 358	6 316	69.19
	vga_lcd	87	17 079	31 501	81.19
	wb_conmax	1 130	770	15 810	57.64
NXP	p35k	742	2 173	8 591	65.18
	p45k	1 411	2 331	11 413	46.74
	p78k	174	2 977	25 740	46.57
	p81k	155	3 877	44 559	34.16
	p89k	406	4 225	25 209	48.13
	p100k	170	5 395	25 633	49.33
	p267k	807	14 628	47 986	57.35
	p295k	46	16 358	52 366	61.10
	p330k	1 238	11 946	54 287	44.43
	p378k	850	14 885	125 824	46.98
	p388k	1 219	19 367	118 920	56.36
	p533k	962	28 676	185 652	55.26
AES	2-2-2-8_d	64	0	5 597	34.16
	2-2-2-8_e	64	0	4 887	39.14
	2-4-4-4_d	128	0	2 056	13.28
	2-4-4-4_e	128	0	1 726	7.13
	10-2-2-4_d	32	0	1 923	18.82
	10-2-2-4_e	32	0	1 936	10.18
	10-2-4-4_d	64	0	3 462	19.64
	10-2-4-4_e	64	0	3 515	9.59

B. Complete List of Publications by the Author

Journal Articles

[J1] J. Burchard, D. Erb, S. M. Reddy, A. D. Singh, and B. Becker, “On the generation of waveform-accurate hazard and charge-sharing aware tests for transistor stuck-off faults in CMOS logic circuits”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2017. DOI: 10.1109/TCAD.2017.2772825.

[J2] P. Raiola, J. Burchard, F. Neubauer, D. Erb, and B. Becker, “Evaluating the effectiveness of D-chains in SAT-based ATPG and diagnostic TPG”, *Journal of Electronic Testing: Theory and Applications (JETTA)*, 2017. DOI: 10.1007/s10836-017-5693-6.

Conference Papers in Formal Proceedings

[C1] J. Burchard, F. Neubauer, P. Raiola, D. Erb, and B. Becker, “Evaluating the effectiveness of D-chains in SAT-based ATPG”, in *18th IEEE Latin American Test Symposium (LATS)*, 2017. DOI: 10.1109/LATW.2017.7906752.

[C2] J. Burchard, D. Erb, A. D. Singh, S. M. Reddy, and B. Becker, “Fast and waveform-accurate hazard-aware SAT-based TSOF ATPG”, in *Design, Automation and Test in Europe (DATE), 2017*, Best Paper Award in the Test Category, 2017. DOI: 10.23919/DATE.2017.7927027.

[C3] J. Burchard, D. Erb, S. M. Reddy, A. D. Singh, and B. Becker, “Efficient SAT-based generation of hazard-activated TSOF tests”, in *IEEE 35th VLSI Test Symposium (VTS)*, 2017. DOI: 10.1109/VTS.2017.7928943.

[C4] J. Burchard, T. Schubert, and B. Becker, “Laissez-faire caching for parallel #SAT solving”, in *SAT 2015*, ser. Lecture Notes in Computer Science, Springer International Publishing, 2015. DOI: 10.1007/978-3-319-24318-4_5.

[C5] J. Burchard, T. Schubert, and B. Becker, “Distributed parallel #SAT solving”, in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2016. DOI: 10.1109/CLUSTER.2016.20.

[C6] J. Burchard, D. Erb, and B. Becker, “Characterization of possibly detected faults by accurately computing their detection probability”, in *Design, Automation and Test in Europe (DATE), 2018*, 2018. DOI: 10.23919/DATE.2018.8342040.

[C7] T. Schubert, J. Burchard, M. Sauer, and B. Becker, “S-trike: a mobile robot platform for higher education”, in *International Conference on Computer Applications in Industry and Engineering*, 2013.

[C8] J. Horáček, J. Burchard, B. Becker, and M. Kreuzer, “Integrating algebraic and SAT solvers”, in *International Conference on Mathematical Aspects of Computer and Information Sciences (MACIS)*, 2017. DOI: 10.1007/978-3-319-72453-9_11.

Workshop Articles

[W1] M. Sauer, J. Burchard, T. Schubert, I. Polian, and B. Becker, “Waveform-guided fault injection by clock manipulation”, in *TRUDEVICE Workshop*, 2013.

[W2] M. Gay, J. Burchard, J. Horáček, A. M. Ekossono, T. Schubert, B. Becker, I. Polian, and M. Kreuzer, “Small scale AES toolbox: algebraic and propositional formulas, circuit-implementations and fault equations”, in *FCTRU*, 2016.

[W3] J. Burchard, A. M. Ekossono, J. Horáček, T. Schubert, M. Gay, B. Becker, M. Kreuzer, and I. Polian, “Towards mixed structural-functional models for algebraic fault attacks on ciphers”, in *International Verification and Security Workshop (IVSW)*, 2017.

[W4] ——, “Towards mixed structural-functional models for algebraic fault attacks on ciphers”, in *RESCUE Workshop on Reliability, Security and Quality at ETS*, 2017.

[W5] J. Burchard, M. Gay, A. M. Ekossono, J. Horáček, T. Schubert, B. Becker, M. Kreuzer, and I. Polian, “AutoFault: towards automatic construction of algebraic fault attacks”, in *Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2017.

List of Figures

List of Tables

Bibliography

[1] A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications.* Amsterdam, The Netherlands: IOS Press, 2009, ISBN: 1586039296, 9781586039295.

[2] H.-J. Wunderlich, *Models in Hardware Testing: Lecture Notes of the Forum in Honor of Christian Landrault*, 1st. Springer Publishing Company, Incorporated, 2009, ISBN: 9048132819, 9789048132812.

[3] S. A. Cook, "The complexity of theorem-proving procedures", in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, ser. STOC '71, Shaker Heights, Ohio, USA: ACM. DOI: 10.1145/800157.805047.

[4] G. Tseitin, "On the Complexity of Derivation in Propositional Calculus", *Studies in Constructive Mathematics and Mathematical Logic*, 1968.

[5] B. Selman, H. Levesque, and D. Mitchell, "A new method for solving hard satisfiability problems", in *Proceedings of the Tenth National Conference on Artificial Intelligence*, ser. AAAI'92, 1992, ISBN: 0-262-51063-4.

[6] B. Selman, H. A. Kautz, and B. Cohen, "Local search strategies for satisfiability testing", in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1996.

[7] M. Davis and H. Putnam, "A computing procedure for quantification theory", *J. ACM*, vol. 7, no. 3, pp. 201–215, Jul. 1960. DOI: 10.1145/321033.321034.

[8] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving", *Commun. ACM*, 1962. DOI: 10.1145/368273.368557.

[9] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: engineering an efficient SAT solver", in *38th Design Automation Conference (DAC)*, 2001. DOI: 10.1145/378239.379017.

[10] N. Eén and N. Sörensson, "An extensible SAT-solver", in *Theory and Applications of Satisfiability Testing (SAT)*, 2004. DOI: 10.1007/978-3-540-24605-3_37.

[11] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination", in *Theory and Applications of Satisfiability Testing (SAT)*, F. Bacchus and T. Walsh, Eds., 2005. DOI: 10.1007/11499107_5.

[12] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems", in *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, Swansea, UK, 2009. DOI: 10.1007/978-3-642-02777-2_24.

[13] T. Schubert, M. Lewis, and B. Becker, "Antom – solver description", SAT Race, 2010.

[14] T. Schubert and S. Reimer, “Antom”, in *https://projects.informatik.uni-freiburg.de/projects/antom*, 2016.

[15] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, “Symbolic model checking without BDDs”, in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 1999. DOI: 10.1007/3-540-49059-0_14.

[16] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990, ISBN: 0716710455.

[17] R. H. Katz, *Contemporary Logic Design*, 1st. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1994, ISBN: 0805327134.

[18] H. T. Nagle, S. C. Roy, C. F. Hawkins, M. G. McNamer, and R. R. Fritzemeier, “Design for testability and built-in self test: a review”, *IEEE Transactions on Industrial Electronics*, vol. 36, no. 2, 1989. DOI: 10.1109/41.19062.

[19] M. K. Reddy and S. M. Reddy, “Detecting FET stuck-open faults in CMOS latches and flip-flops”, *IEEE Design Test of Computers*, vol. 3, no. 5, 1986. DOI: 10.1109/MDT.1986.295040.

[20] S. R. Maka and E. J. McCluskey, “ATPG for scan chain latches and flip-flops”, in *15th IEEE VLSI Test Symposium*, 1997. DOI: 10.1109/VTEST.1997.600306.

[21] F. Yang, S. Chakravarty, N. Devta-Prasanna, S. M. Reddy, and I. Pomeranz, “Detection of internal stuck-open faults in scan chains”, in *IEEE International Test Conference*, 2008. DOI: 10.1109/TEST.2008.4700577.

[22] J. Savir and S. Patil, “Broad-side delay test”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 8, 1994.

[23] J. Saxena, K. M. Butler, J. Gatt, R. Raghuraman, S. P. Kumar, S. Basu, D. J. Campbell, and J. Berech, “Scan-based transition fault testing - implementation and low cost test challenges”, in *Proceedings. International Test Conference*, 2002. DOI: 10.1109/TEST.2002.1041869.

[24] S. Dasgupta, R. G. Walther, T. W Williams, and E. B. Eichelberger, “An enhancement to lssd and some applications of lssd in reliability, availability and serviceability”, in *Symposium on Fault-Tolerant Computing*, 1981.

[25] F. M. Wanlass, “Low stand-by power complementary field effect circuitry”, pat. 3,356,858, 1963.

[26] Si2, *NanGate FreePDK45 generic open cell library, v1.3*, http://www.si2.org/openeda.si2.org/projects/nangatelib.

[27] J. M. Galey, R. E. Norby, and J. P. Roth, “Techniques for the diagnosis of switching circuit failures”, in *2nd Annual Symposium on Switching Circuit Theory and Logical Design (SWCT)*, 1961. DOI: 10.1109/FOCS.1961.33.

[28] E. R. Hsieh, R. A. Rasmussen, L. J. Vidunas, and W. T. Davis, “Delay test generation”, in *Proceedings of the 14th Design Automation Conference*, ser. DAC ’77, IEEE Press, 1977.

[29] R. L. Wadsack, “Fault modeling and logic simulation of CMOS and MOS integrated circuits”, *Bell System Technical Journal*, vol. 57, no. 5, 1978. DOI: 10.1002/j.1538-7305.1978.tb02106.x.

[30] G. L. Smith, “Model for delay faults based upon paths”, in *International Test Conference*, 1985.

[31] I. Pomeranz and S. M. Reddy, “Transition path delay faults: a new path delay fault model for small and large delay defects”, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, 2008. DOI: 10.1109/TVLSI.2007.909796.

[32] S. M. Reddy, “Models for delay faults”, in *Models in Hardware Testing*, H.-J. Wunderlich, Ed. Springer Netherlands, 2010, pp. 71–103. DOI: 10.1007/978-90-481-3282-9_3.

[33] F. Hapke, W. Redemund, A. Glowatz, J. Rajski, M. Reese, M. Hustava, M. Keim, J. Schloeffel, and A. Fast, “Cell-aware test”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 9, 2014. DOI: 10.1109/TCAD.2014.2323216.

[34] F. Hapke, M. Reese, J. Rivers, A. Over, V. Ravikumar, W. Redemund, A. Glowatz, J. Schloeffel, and J. Rajski, “Cell-aware production test results from a 32-nm notebook processor”, in *IEEE International Test Conference*, 2012. DOI: 10.1109/TEST.2012.6401533.

[35] F. Hapke, R. Arnold, M. Beck, *et al.*, “Cell-aware experiences in a high-quality automotive test suite”, in *IEEE European Test Symposium (ETS)*, 2014. DOI: 10.1109/ETS.2014.6847814.

[36] J. P. Roth, “Diagnosis of automata failures: a calculus and a method”, *IBM Journal of Research and Development*, vol. 10, no. 4, 1966. DOI: 10.1147/rd.104.0278.

[37] P. Goel, “An implicit enumeration algorithm to generate tests for combinational logic circuits”, *IEEE Transactions on Computers*, vol. C-30, no. 3, 1981. DOI: 10.1109/TC.1981.1675757.

[38] H. Fujiwara and T. Shimono, “On the acceleration of test generation algorithms”, *IEEE Transactions on Computers*, vol. C-32, no. 12, 1983. DOI: 10.1109/TC.1983.1676174.

[39] T. Larrabee, “Test pattern generation using Boolean satisfiability”, *IEEE Transactions on Computer-Aided Design*, vol. 11, no. 1, 1992. DOI: 10.1109/43.108614.

[40] D. Tille and R. Drechsler, “A fast untestability proof for SAT-based ATPG”, in *12th International Symposium on Design and Diagnostics of Electronic Circuits Systems*, 2009. DOI: 10.1109/DDECS.2009.5012096.

[41] J. Shi, G. Fey, R. Drechsler, A. Glowatz, J. Schloffel, and F. Hapke, “Experimental studies on SAT-based test pattern generation for industrial circuits”, in *6th International Conference on ASIC*, vol. 2, 2005. DOI: 10.1109/ICASIC.2005.1611489.

[42] P. Tafertshofer and A. Ganz, “SAT based ATPG using fast justification and propagation in the implication graph”, in *IEEE/ACM International Conference on Computer-Aided Design*, 1999. DOI: 10.1109/ICCAD.1999.810638.

[43] S. Eggersglüß, R. Krenz-Bååth, A. Glowatz, F. Hapke, and R. Drechsler, "A new SAT-based ATPG for generating highly compacted test sets", in *IEEE 15th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2012. DOI: `10.1109/DDECS.2012.6219063`.

[44] M. Sauer, B. Becker, and I. Polian, "PHAETON: a SAT-based framework for timing-aware path sensitization", *IEEE Transactions on Computers*, vol. 65, no. 6, 2016. DOI: `10.1109/TC.2015.2458869`.

[45] D. Erb, K. Scheibler, M. A. Kochte, M. Sauer, H. J. Wunderlich, and B. Becker, "Mixed 01X-RSL-encoding for fast and accurate ATPG with unknowns", in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2016. DOI: `10.1109/ASPDAC.2016.7428101`.

[46] K. Scheibler, D. Erb, and B. Becker, "Accurate CEGAR-based ATPG in presence of unknown values for large industrial designs", in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2016, ISBN: 978-3-9815370-6-2.

[47] D. Tille and R. Drechsler, "Incremental SAT instance generation for SAT-based ATPG", in *2008 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, 2008. DOI: `10.1109/DDECS.2008.4538759`.

[48] M. Sauer, A. Czutro, I. Polian, and B. Becker, "Small-delay-fault ATPG with waveform accuracy", in *Int'l Conf. on CAD*, 2012.

[49] D. Brand, "Verification of large synthesized designs", in *International Conference on Computer Aided Design (ICCAD)*, 1993. DOI: `10.1109/ICCAD.1993.580110`.

[50] G. Fey, J. Shi, and R. Drechsler, "Efficiency of multi-valued encoding in SAT-based ATPG", in *36th International Symposium on Multiple-Valued Logic (ISMVL'06)*, 2006. DOI: `10.1109/ISMVL.2006.19`.

[51] K. Yang, K. T. Cheng, and L. C. Wang, "Trangen: a SAT-based ATPG for path-oriented transition faults", in *ASP-DAC: Asia and South Pacific Design Automation Conference*, 2004, pp. 92–97.

[52] H. Chen and J. Marques-Silva, "Tg-pro: a new model for SAT-based ATPG", in *IEEE International High Level Design Validation and Test Workshop*, 2009, pp. 76–81.

[53] D. Erb, K. Scheibler, M. Sauer, and B. Becker, "Efficient SMT-based ATPG for interconnect open defects", in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2014. DOI: `10.7873/DATE.2014.138`.

[54] P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Combinational test generation using satisfiability", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 9, 1996. DOI: `10.1109/43.536723`.

[55] D. Erb, K. Scheibler, M. A. Kochte, M. Sauer, H. J. Wunderlich, and B. Becker, "Test pattern generation in presence of unknown values based on restricted symbolic logic", in *ITC*, 2014. DOI: `10.1109/TEST.2014.7035350`.

[56] H. Chen and J. Marques-Silva, "A two-variable model for SAT-based ATPG", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 12, 2013. DOI: `10.1109/TCAD.2013.2275254`.

[57] A. Riefert, *Test and Diagnosis of Embedded Processor Cores with Formal Methods.* Der Andere Verlag, 2016.

[58] I. Hamzaoglu and J. H. Patel, "New techniques for deterministic test pattern generation", *Journal of Electronic Testing*, vol. 15, no. 1, 1999. DOI: `10.1023/A:1008355411566`.

[59] K. H. Rosen, *Discrete Mathematics and Its Applications*, 5th. McGraw-Hill Higher Education, 2002, ISBN: 0072424346.

[60] N. Devtaprasanna, A. Gunda, P. Krishnamurthy, S. M. Reddy, and I. Pomeranz, "A unified method to detect transistor stuck-open faults and transition delay faults", in *IEEE European Test Symposium*, 2006.

[61] S. K. Jain and V. D. Agrawal, "Test generation for MOS circuits using D-algorithm", in *Design Automation Conference*, ser. DAC '83, Miami Beach, Florida, USA: IEEE Press, 1983, ISBN: 0-8186-0026-8.

[62] S. M. Reddy, M. K. Reddy, and J. G. Kuhl, "On testable design for CMOS logic circuits.", in *International Test Conference*, Philadelphia, PA, USA, 1983.

[63] D. L. Liu and E. J. McCluskey, "Designing CMOS circuits for switch-level testability", *IEEE Design Test of Computers*, vol. 4, no. 4, 1987.

[64] K. J. Lee and M. A. Breuer, "On the charge sharing problem in CMOS stuck-open fault testing", in *IEEE International Test Conference*, 1990.

[65] N. K. Jha and J. A. Abraham, "Design of testable CMOS logic circuits under arbitrary delays", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 4, no. 3, 1985. DOI: `10.1109/TCAD.1985.1270122`.

[66] S. Kundu and S. M. Reddy, "On the design of robust testable CMOS combinational logic circuits", in *Fault-Tolerant Computing, FTCS-18*, 1988.

[67] S. M. Reddy and M. K. Reddy, "Testable realizations for FET stuck-open faults in CMOS combinational logic circuits", *IEEE Transactions on Computers*, vol. C-35, no. 8, 1986.

[68] S. Chakravarty, "A testable realization of CMOS combinational circuits", in *International Test Conference*, 1989.

[69] S. M. Reddy, M. K. Reddy, and V. Agrawal, "Robust tests for stuck-open faults in CMOS combinational logic circuits", in *Fault-Tolerant Computing, FTCS-14*, 1984.

[70] M. Yoeli and S. Rinon, "Application of ternary algebra to the study of static hazards", *J. ACM*, 1964.

[71] I. Pomeranz and S. M. Reddy, "Hazard-based detection conditions for improved transition path delay fault coverage", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 9, 2010. DOI: `10.1109/TCAD.2010.2049462`.

[72] ——, "Hazard-based detection conditions for improved transition fault coverage of scan-based tests", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 2, 2010. DOI: `10.1109/TVLSI.2008.2010216`.

[73] I. Pomeranz, “Static test compaction for transition faults under the hazard-based detection conditions”, in *IEEE 30th VLSI Test Symposium (VTS)*, 2012. DOI: 10.1109/VTS.2012.6231099.

[74] W. Shockley, “Problems related to p-n junctions in silicon”, *Solid-State Electronics*, vol. 2, no. 1, 1961. DOI: 10.1016/0038-1101(61)90054-5.

[75] W. Schemmert and G. Zimmer, “Threshold-voltage sensitivity of ion-implanted m.o.s. transistors due to process variations”, *Electronics Letters*, vol. 10, no. 9, 1974. DOI: 10.1049/el:19740115.

[76] B. Kempf, “The boost.threads library”, *C/C++ Users Journal*, 2002.

[77] T. Sang, F. Bacchus, P. Beame, H. A. Kautz, and T. Pitassi, “Combining component caching and clause learning for effective model counting”, in *SAT 2004*, 2004.

[78] Message Passing Forum, “MPI: a message-passing interface standard”, Knoxville, TN, USA, Tech. Rep., 1994.

[79] T. Schubert, M. D. T. Lewis, and B. Becker, “PaMiraXT: parallel SAT solving with threads and message passing”, *JSAT*, vol. 6, no. 4, 2009.

[80] M. Thurley, “sharpSAT: counting models with advanced component caching and implicit BCP”, in *SAT 2006*, 2006.

[81] S. Chakraborty, K. S. Meel, and M. Y. Vardi, “A scalable approximate model counter”, in *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming - Volume 8124*, Uppsala, Sweden: Springer-Verlag New York, Inc., 2013. DOI: 10.1007/978-3-642-40627-0_18.

[82] ——, “Algorithmic improvements in approximate counting for probabilistic inference: from linear to logarithmic SAT calls”, in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI)*, New York, New York, USA, 2016, ISBN: 978-1-57735-770-4.

[83] M. Böhm and E. Speckenmeyer, “A fast parallel SAT-solver — efficient workload balancing”, *Annals of Mathematics and Artificial Intelligence*, vol. 17, no. 2, 1996. DOI: 10.1007/BF02127976.

[84] Y. Hamadi, S. Jabbour, and L. Sais, “ManySAT: a parallel SAT solver”, *JSAT*, vol. 6, no. 4, 2009.

[85] W. Chrabakh and R. Wolski, “GridSAT: a chaff-based distributed SAT solver for the grid”, in *Supercomputing, 2003 ACM/IEEE Conference*, 2003. DOI: 10.1145/1048935.1050188.

[86] Y. Hamadi, S. Jabbour, and L. Sais, “Manysat: a parallel sat solver”, *JSAT*, vol. 6, 2009.

[87] M. Kaufmann, S. Kottler, M. Kaufmann, and S. Kottler, “SArTagnan - a parallel portfolio SAT solver with lockless physical clause sharing”, in *In Pragmatics of SAT*, 2011.

[88] T. Balyo, P. Sanders, and C. Sinz, “Hordesat: A massively parallel portfolio SAT solver”, *CoRR*, vol. abs/1505.03340, 2015.

[89] I. Pomeranz, L. N. Reddy, and S. M. Reddy, "Compactest: a method to generate compact test sets for combinational circuits", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 7, 1993. DOI: 10.1109/43.238040.

[90] N. Zacharia, J. Rajski, and J. Tyszer, "Decompression of test data using variable-length seed LFSRs", in *Proceedings 13th IEEE VLSI Test Symposium*, 1995. DOI: 10.1109/VTEST.1995.512670.

[91] H. Cho, S.-W. Jeong, F. Somenzi, and C. Pixley, "Synchronizing sequences and symbolic traversal techniques in test generation", *Journal of Electronic Testing*, vol. 4, no. 1, 1993. DOI: 10.1007/BF00971937.

[92] I. Pomeranz and S. M. Reddy, "On synchronizable circuits and their synchronizing sequences", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 9, 2000. DOI: 10.1109/43.863649.

[93] M. Keim, B. Becker, and B. Stenner, "On the (non-)resetability of synchronous sequential circuits", in *Proceedings of 14th VLSI Test Symposium*, 1996. DOI: 10.1109/VTEST.1996.510863.

[94] D. Erb, M. A. Kochte, S. Reimer, M. Sauer, H. J. Wunderlich, and B. Becker, "Accurate QBF-based test pattern generation in presence of unknown values", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 12, 2015.

[95] M. Elm, M. A. Kochte, and H.-J. Wunderlich, "On determining the real output Xs by SAT-based reasoning", in *Proc. IEEE Asian Test Symposium*, 2010. DOI: 10.1109/ATS.2010.16.

[96] H.-Z. Chou, K.-H. Chang, and S.-Y. Kuo, "Accurately handle don't-care conditions in high-level designs and application for reducing initialized registers", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 4, 2010. DOI: 10.1109/TCAD.2010.2042905.

[97] *Tessent shell reference manual*, version Software Version 2016.2, Mentor Graphics Corporation, June 2016.

[98] *TetraMAX ATPG quick reference*, version Version A-2007.12, Synopsys, December 2007.

[99] M. A. Kochte and H. J. Wunderlich, "SAT-based fault coverage evaluation in the presence of unknown values", in *2011 Design, Automation Test in Europe*, 2011. DOI: 10.1109/DATE.2011.5763209.

[100] F. Corno, M. S. Reorda, and G. Squillero, "RT-level ITC'99 benchmarks and first ATPG results", *IEEE Des. Test*, vol. 17, no. 3, 2000. DOI: 10.1109/54.867894.

[101] C. Albrecht, "IWLS 2005 benchmarks", in *International Workshop on Logic Synthesis*, Jun. 2005.