Stellingen

behorende bij het proefschrift

Automatic Test Pattern Generation for
Synchronous Sequential Circuits

Marinus Hendrik Konijnenburg

Delft, 21 december 1998
1. De toenemende noodzaak voor het ontwerpen van goedkopere geïntegreerde schakelingen staat op gespannen voet met het vaak als vanzelfsprekend ervaren default gebruik van scan-flipflops.

2. Het ontwikkelen van nieuwe zoekstrategieën voor test generatie voor digitale schakelingen is niet zo moeilijk. De kunst is het bepalen van de juiste strategieën voor een schakeling. Net als in het dagelijks leven.

3. Research and development in chip testing is stuck-at the stuck-at fault model.

4. The feedback loops within sequential circuits usually reduce the circuit’s testability. The feedback “loop” from the test engineer to the designers of sequential circuits usually improves the circuit’s testability.

5. Er is een balans tussen CPU gebruik en geheugen gebruik bij het ontwikkelen van algoritmen voor NP-complete problemen, zoals ATPG: een verminderd CPU gebruik leidt tot een verhoogd geheugen gebruik en vice versa.

6. Een nadeel van de backtrack algoritmen is dat ze een willekeurige beslissing herzien in geval van een conflict, namelijk de laatste beslissing.

7. Het cliché van de jaren negentig: “Ik heb het druk”.

8. De veiligheid op straat evenals het milieu zijn er ten zeerste mee gebaat als er een tarief per auto-rit ingesteld wordt. Juist de zeer inefficiënte en milieu onvriendelijke korte auto-rit wordt hiermee beperkt, wat de veiligheid in steden en dorpen ten goede komt.

9. De 10%-90% regel uit de statistiek is vaak ook van toepassing voor vergaderingen: 90% van de totale produktiviteit is verricht binnen 10% van de vergadertijd.


11. Hoewel niet alle hedendaagse ethische vraagstukken in onze samenleving rechtstreeks door de 10 geboden beantwoord kunnen worden, toch bevatten de 10 geboden de fundamentele richtlijnen voor ons handelen met God en onze medemens.


13. Tegenwoordig bepaalt de publieke opinie welke normen en waarden er gelden. Hierdoor vervaagt de moraliteit.


15. DAT is het pakket waar ik al vele jaren met plezier aan werk.
1. The increasing necessity to design integrated circuits at lower costs is at odds with the default application of scan-flipflops which is often taken for granted.

2. Development of new search-strategies for test generation of digital circuits is not that hard. The art is to determine the right strategies for a circuit. Just like in real life.

3. Research and development in chip testing is stuck-at the stuck-at fault model.

4. The feedback loops within sequential circuits usually reduce the circuit's testability. The feedback "loop" from the test engineer to the designers of sequential circuits usually improves the circuit's testability.

5. There is a balance between CPU usage and memory usage, when developing software for NP-complete problems, such as ATPG: A reduced CPU usage results in an increased memory usage and vice versa.

6. A disadvantage of the backtrack algorithms is that they remake an arbitrary decision in case of a conflict, namely the last decision.

7. The cliché of the nineties: "I am busy"

8. The safety on the streets as well as the environment would benefit greatly when taxes per car-ride are introduced. Especially the number of very inefficient and environment-unfriendly short car-rides will be reduced, improving the safety in cities and villages.

9. The 10%-90% rule from statistics can often be applied on meetings: 90% of the productivity has been performed within 10% of the consumed meeting time.

10. A software-bug is often identified by yourself, when you explain the software's expected behavior to others. When teaching, often a teacher also learns himself.

11. Although not all current ethical problems in our society can be solved using the Ten Commandments, still they contain the fundamental guidelines for our behavior to God and to our fellow human-beings.

12. Sentiments and emotions are capable to undermine a healthy and stable economic system.

13. Nowadays, the public opinion determines standards and values. Because of this morality is fading.

14. Where would we be without trains: in a traffic-jam.

15. DAT is the package I worked on with great pleasure, for many years now.
Automatic Test Pattern Generation for Synchronous Sequential Circuits
Automatic Test Pattern Generation for Synchronous Sequential Circuits

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof.ir. K.F. Wakker,
in het openbaar te verdedigen ten overstaan van een commissie,
door het College van Promoties aangewezen,

op maandag 21 december 1998 te 16:00 uur
door Marinus Hendrik KONIJNENBURG
elektrotechnisch ingenieur
tegen te Ridderkerk
Dit proefschrift is goedgekeurd door de promotor:

Prof.dr.ir. A.J. van de Goor

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter
Prof.dr.ir. A.J. van de Goor, promotor
Prof.dr.ir. L. Claesen
Prof.dr.ir. P.M. Dewilde
Prof.ir. M.T.M. Segers
Prof.dr. H. Wallinga
K. Baker M.Sc.
Dipl.-Ing. F. Hapke

Technische Universiteit Delft
Technische Universiteit Delft
IMEC, Leuven, België
Technische Universiteit Delft
Technische Universiteit Eindhoven
Universiteit Twente
Philips Research, Eindhoven
Philips Semiconductors, Hamburg, Duitsland

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG
Konijnenburg, Marinus Hendrik
Automatic Test Pattern Generation for Synchronous Sequential Circuits
/ Marinus Hendrik Konijnenburg.
PhD. Thesis Technische Universiteit Delft. - With ref. - With summary in Dutch.
ISBN 90-9012096-3
Subject headings: Digital systems testing / automatic test pattern generation.

Copyright ©1998 by Mario Konijnenburg

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior permission of the author.
Aan Marianne en aan Janine en Erica.
# Contents

Abstract v  
Preface and acknowledgements vii  

1 Introduction 1  
1.1 Background: Test types and logic circuit types 2  
1.2 Complexity of TPG for sequential circuits 4  
1.3 Design for testability and its limitations 6  
1.4 Scope of this thesis 8  
1.5 Organization of this thesis 9  

2 Circuit, signal, and fault models 13  
2.1 The combinational circuit model 14  
2.2 The sequential circuit model 16  
2.2.1 Latches and flipflops 17  
2.2.2 Finite state machines 19  
2.2.3 Circuit state initialization 21  
2.3 Iterative array model of sequential circuits 24  
2.4 The 5-valued signal model for 3-state circuits 24  
2.5 The stuck-at fault model 26  
2.5.1 Classification of fault models 26  
2.5.2 Stuck-at fault detection condition 27  
2.5.3 Classification of fault instances according to the detectability condition 29  
2.6 Summary and conclusions 31  

3 ATPG system structure 33  
3.1 Main components of an ATPG system 34  
3.2 DTPG for combinational circuits 39  
3.2.1 The FAN algorithm for combinational circuits 39  
3.2.2 DAT: Test generation for 3-state combinational circuits 44  
3.3 DTPG for sequential circuits 46  
3.3.1 Time observation approaches 47  
3.3.2 Time processing directions in TPG 49  
3.4 Summary and conclusions 53
## 4 Problems in STPG and proposed solutions

4.1 The problems ............................................. 56
   4.1.1 Bus-conflict detection and the applied signal model ...... 56
   4.1.2 Complete and efficient search and the applied signal model ... 57
   4.1.3 Illegal state space detection and over-specified ISes .......... 60
4.2 Prior art in STPG, related to signal models .................. 61
4.3 Prior art in STPG, related to illegal state detection .......... 65
4.4 Evaluation of STPG methods ................................... 66
4.5 Open questions and proposed answers ............................. 70
4.6 Summary and conclusions ....................................... 73

## 5 ATPG for sequential circuits

5.1 Pre-processes .................................................. 76
   5.1.1 Can-be-Z and can-be-U calculation .......................... 76
   5.1.2 Controllability and observability cost calculations .......... 76
   5.1.3 Identification of untestable faults prior to TPG .............. 80
5.2 Deterministic test generation ................................... 83
   5.2.1 Three phases during STPG: Activation, propagation and justifica-
        tion .................................................. 84
   5.2.2 Overview of the STPG algorithm ................................ 85
   5.2.3 Fault dependent cost estimates ................................ 88
   5.2.4 Fault propagation .......................................... 91
   5.2.5 Backtrace with objectives having six counters ................ 94
   5.2.6 Three-valued decisions and decision stacks .................... 94
   5.2.7 Internal restrictions .......................................... 95
   5.2.8 Backtrack procedure ......................................... 101
5.3 Test generation strategies ....................................... 103
   5.3.1 Fault propagation before justification of objectives .......... 104
   5.3.2 Order of search during backtrace ............................. 104
   5.3.3 Single backtrace ............................................. 105
   5.3.4 Targeted D-front propagation .................................. 105
   5.3.5 Find PPI state in previous generated test sequence .......... 106
   5.3.6 Ignore PPO reached, when PO is reachable in timeframe ........ 106
   5.3.7 State justification before fault propagation ................... 107
   5.3.8 Recursive learning ............................................ 108
5.4 Static learning to support STPG .................................. 109
   5.4.1 Static learning based on logic implication ..................... 110
   5.4.2 Static learning based on signal line justification ............. 115
5.5 Fault simulation .................................................. 116
5.6 Additional target test generation ................................... 117
5.7 Overview of the ATPG process ..................................... 121
5.8 Summary and conclusions .......................................... 123
6 ATPG based on Power-Set Logic .................................................. 125
  6.1 PSL: A power-set based signal model ................................. 126
     6.1.1 Description and notation of PSL ................................. 126
     6.1.2 Default values of signal lines ................................. 128
  6.2 PSL compared to other signal models .................................. 129
  6.3 PSL and TPG for combinational 3-state circuits .................. 131
  6.4 PSL and TPG for sequential circuits ................................. 135
  6.5 PSL implementation aspects ............................................ 137
  6.6 Experimental results ................................................... 138
  6.7 Summary and conclusions .............................................. 141

7 Illegal state space identification ........................................... 143
  7.1 Introduction to global illegal states ............................... 144
  7.2 Static learning based on state justification to identify GISes .. 146
  7.3 Legal state cache ...................................................... 147
  7.4 Storage of (G)ISes .................................................... 150
      7.4.1 Groups of flipflops (FF-groups) ............................... 151
      7.4.2 Compressed vectors ............................................ 152
  7.5 Expanding the known GIS space ....................................... 152
  7.6 Experimental results .................................................. 161
  7.7 Summary and conclusions .............................................. 165

8 Case study: ATPG for the Philips 80C51 .................................. 171
  8.1 Introduction ........................................................... 172
  8.2 Circuit description .................................................... 173
  8.3 Test generation in full-scan mode ................................... 175
  8.4 Test generation in non-scan mode .................................... 176
  8.5 Test generation in partial-scan mode ................................ 178
      8.5.1 Scan flipflop selection ........................................ 179
      8.5.2 Scan flipflop reduction ....................................... 182
      8.5.3 Experimental results ......................................... 185
  8.6 Summary and conclusions .............................................. 188

9 Conclusions .......................................................................... 191
  9.1 Major contributions ..................................................... 195
  9.2 Suggestions for future research ....................................... 196

Bibliography ............................................................................ 198

A ISCAS '85 and '89 circuits .................................................... 209

B Industrial circuit designs ..................................................... 211

C DAT: Delft Advanced Test generation tool .............................. 215

Samenvatting (summary in Dutch) ............................................. 217
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Curriculum vitae</td>
<td>219</td>
</tr>
<tr>
<td>Index</td>
<td>221</td>
</tr>
</tbody>
</table>
Abstract

Automatic Test Pattern Generation for Synchronous Sequential Circuits

The main subject of this thesis is Automatic Test Pattern Generation (ATPG) for synchronous sequential, Boolean and three-state circuits using the single stuck-at fault model. Sequential circuits have the capability to store information from the past into memory, which forms the state of the circuit.

The main problem of ATPG for sequential circuits (without or with partial scan) is its very high complexity; until now it is by far not solved. This complexity is mainly due to a circuit attribute, termed density of encoding; a low density of encoding means that many states are not functionally used during normal circuit operation. These states are called illegal and often are not justifiable. Another problem is over-specification on the states during ATPG; i.e., states which have assigned entries which do not contribute to the detection of the current fault-under-test. This may result in larger test sets, or even worse, faults being incorrectly marked untestable (i.e., search incompleteness).

The main contributions of this thesis are a new signal model based on power-sets, and techniques to identify, store and “optimize” Illegal States (ISes). The new signal model solves the over-specification problem, thereby assuring search completeness, and is much more effective during test generation than other test generation methods. IS identification and optimization reduce the complexity of ATPG significantly, because no effort is spent in repeatedly trying to justify ISes during test generation. The ATPG system, Delft Advanced Test generation (DAT), has been extended for sequential circuits, by implementing these contributions.

To demonstrate the effectiveness and efficiency of this ATPG system, experiments were performed on the ISCAS'89 benchmark circuits and a number of (3-state) industrial circuits. In addition, a case study has been performed on the Philips 80C51 micro-controller: ATPG on the 80C51 without scan, with full-scan, and with partial-scan. To create partial scan versions of the 80C51, partial-scan selection techniques have been proposed and described.
Preface and acknowledgements

As part of my MSEE requirements at the Delft University of Technology I fulfilled a task assignment for Prof. A.J. van de Goor, consisting of a literature research on Iddq testing. Thereafter, based on the results of this research, I designed and implemented a test generation algorithm for Iddq testing based on the leakage fault model, which was my master’s thesis assignment. This algorithm has been implemented as part of the Delft Advanced Test (DAT) generation system, which was being developed by Dr. Hans van der Linden in 1991 and 1992. At the end of my master’s assignment, in September 1992, Prof. van de Goor and Hans asked me to join the test group as software developer, to assist Hans in the development of the DAT ATPG system.

The project to develop the DAT ATPG system started in 1991, in cooperation with Philips Electronics N.V., funded by one-year contracts. In the years 1991 to 1995, the goal was to design and implement a production-grade, state-of-the-art ATPG system that can handle combinational three-state circuits. Fast test generation of compact test sets, using various fault models was the main requirement for this ATPG system.

My first project within the test group was the implementation of the restrictor, which was one of the Philips’ requirements for DAT to constrain the search space to allowable test vectors. Thereafter, I started to design and implement the FAN-algorithm, initially for Boolean circuits, later also for three-state circuits. Over the years 1992-1995 Hans and I succeeded to design and implement the intended ATPG system, which is now in use as part of the AMSAL tool, by Philips worldwide. The combination of research in three-state test generation and the practical application of the results of this research in the implementation of a production-grade ATPG system, made this work very interesting and challenging.

Due to the application of DFT-techniques, such as the scan-design methodology (scan-FFs), combinational circuit TPG methods can be applied to sequential circuits. However, often due to various reasons, some or none of the FFs are allowed to become scannable. Therefore, sequential circuit TPG methods are required. However, state-of-the-art TPG methods for these circuits often perform poorly. For example, many sequential circuits of the well-known ISCAS’89 benchmark set could (and can) not be successfully handled by ATPG systems.

In 1994 a research proposal for test generation for sequential circuits had been submitted to the Nederlandse organisatie voor Wetenschappelijk Onderzoek (NWO). The goal of this research was to design and implement an ATPG system for (3-state) sequential circuits, based on the successful combinational circuit ATPG system DAT. This proposal was accepted, and in March 1995 I started my PhD research assignment in this area.
The complexity of sequential circuit test generation is much higher than that of combinational circuit test generation; often circuits with fewer than 10000 lines could not be successfully handled by TPGs. This made this research very attractive. Over the last three years Hans and I had many discussions, resulting in many ideas for sequential circuit TPG, often to tackle yet another ISCAS'89 circuit. We have learned that the success of the ideas often is unpredictable due to huge number of circuit- and TPG parameters: Promising ideas appeared to be useless and ideas that we (or one of us) gave little chance appeared to be very useful. The results of this research are presented in this thesis.

I would like to thank all the people who have contributed to this work. In particular, I like to mention the following contributors: First of all I wish to thank Prof. A.J. van de Goor and Dr. Hans van der Linden. Prof. van de Goor for giving me the opportunity to become a PhD student in his group. I am very grateful to him for his guidance and help during this project, for writing this thesis, and for his advice considering my research project, but also in many other fields. Hans for his unlimited number of ideas, for the many fruitful and motivating discussions we had, and for his critical reading, comments, and corrections on my writings.

Thanks to the involved people of Philips ED&T, especially to Keith Baker, Krijn Kuiper, and Richard Morren for providing many industrial circuits, for giving the opportunity to develop a "real-world" ATPG system, and also for the many fruitful discussions we had. In addition, also thanks for giving me the opportunity to perform a case-study on the Philips 80C51 micro-controller, for which I also like to acknowledge Steven Oostdijk for his guidance and help.

Jeroen Geuzebroek, I like to thank him for his comments on my thesis, and for being a fine colleague. Eyad Sabbah, who wrote the research proposal for NWO, owes my gratitude for the useful conversations we had; Jan Snelders for his support during my research and for taking care of the financial affairs. The former system administrators Jean-Paul van der Jagt and Tobias Nijweide, and the current system administrator Bert Meijls for providing an excellent working computer environment.

I am greatly indebted to my wife Marianne. Especially, in the last year when I locked myself in my office, she was often on her own, taking care of our children Janine and Erica and a lot of things. I like to thank her for her patience and understanding when I was (pre)occupied with my work. Thanks of course, to family and friends for their support and for letting me do other things than my work. Finally, I believe that all of my life is in the hands of God. Therefore, I thank God above all for giving me the strength to complete this research.

Mario Konijnenburg
Zwijndrecht, December 1998
Chapter 1

Introduction

In order to guarantee high quality Integrated Circuits (ICs, "chips"), testing of those ICs is a necessary part of the manufacturing process. The end-product, which is shipped to the customer, usually consists of many ICs. In order to make reliable end-products possible, only a few defective ICs per million are allowed. This necessitates the development of test programs for ICs, which consumes a large part of the IC development and production budget, because each manufactured IC has to be tested, consuming expensive test-time on very expensive Automatic Test Equipment (ATE). The ATE applies the test programs (or tests, for short) to ICs, which have to detect malfunctioning ICs. Hence, it is clear that the tests, which are supplied by the designer and/or test engineer, have to be of high quality. In practice, this means that, because test development is budgeted and test time per IC has to be economically feasible, tests are developed/generated as fast as possible, with a minimal test application time, whereby the manufacturer allows a maximum DPM\(^1\) level. Therefore, much research has been performed in Automatic Test Program / Pattern Generation (ATPG) for ICs to optimize both test-time and fault coverage (which translates into a DPM level).

In this chapter the reader is introduced to the subject of this thesis: **ATPG for (digital) sequential circuits**. Section 1.1 classifies test types and logic circuit types, placing the subject in its context. Section 1.2 outlines the complexity of ATPG for sequential circuits which is illustrated with some ATPG results, showing that our subject is by far not solved. This is the motivation for our research. Section 1.3 describes the concept of **Design for Testability** (DFT), which reduces the complexity of our subject significantly. However, we show that DFT has its disadvantages and limitations. Thus, the motivation for ATPG for sequential circuits still stands. Section 1.4 defines the scope of the research. Section 1.5 presents the organization of this thesis.

\(^1\)Defects per million; i.e., defective parts per million shipped instances
1.1 Background: Test types and logic circuit types

Many tests are used in the production of ICs, as there are many aspects which have to be tested. In general, these tests can be classified into the following four types:

1. Application-mode tests. They test the ICs within their environment.

2. Functional tests. They test the functionality of the ICs: "Does the IC (functionally) do what it is supposed to do?"

3. Structural tests. They test whether the IC is (structurally) built as specified and whether any defects were introduced during its manufacturing process.

4. Parametric tests. They test if the IC is working properly under various parameter conditions, such as operation frequency, dissipation, power supply voltage, and temperature range. These tests are used in combination with application-mode, functional and structural tests.

Application-mode testing concentrates on the board level, and is usually performed by the customer. Therefore, it is outside the scope of this research, because this thesis handles testing during production of the IC.

Functional tests are usually provided by the designer, who knows best what the IC is supposed to do. The disadvantage of functional tests is the huge number of tests which are often required.

The research reported in this thesis refers to generating structural tests for ICs. Those are applied to a manufactured IC to check if the IC is free of defects. Defects, such as undesired shorts, opens and contaminations, are caused for instance by dust or other particles during the manufacturing process, and can change the function of the IC. They appear at the physical level. A fault models the (logic) behavior of one or more physical defects in a circuit\(^2\); e.g., the single stuck-at fault (SAF) model assumes that each signal line in a circuit can become stuck at a fixed Boolean value; i.e., stuck-at-0 (SA-0) and stuck-at-1 (SA-1).

\(^2\)How well a selected set of fault models covers the (behavior of) actual defects, is beyond the scope of this thesis
An IC or circuit usually consists of various blocks. Each type of block has its own typical defects, and has its own test methods. These blocks can be categorized as memory blocks (RAM, ROM), and logic blocks (Boolean gates, three-state (3-state) elements, flipflops), as shown in Figure 1.1. Stimuli are applied to the Primary Inputs (PIs) and responses are captured from the Primary Outputs (POs). The memory block, which has a regular structure, is a memory array consisting of n by m bits of storage. It is tested, for example, by means of march tests [vdG98]. The logic block can be categorized into two types: combinational logic and sequential logic.

A combinational (logic) circuit consists of an interconnected set of gates, containing no feedback loops\(^3\). The output values (on the POs) of a combinational circuit at any given time, depend only on the current (after some delay) input values (on the PIs). Figure 1.2a shows a model of a combinational circuit. The combinational logic circuit is tested by means of a test set; which is a set of test vectors. A test vector consists of a PI stimulus with corresponding PO response. Normally, it is infeasible to apply all possible stimuli \(2^n \ (n = \#\text{PIs})\). Therefore, test generation algorithms are developed which select an appropriate set of vectors. A test set is generated so that each fault is covered by at least

\(^3\)A feedback loop is a directed path from the output of a gate G to an input of that gate G.
one test vector. Figure 1.3a shows an example circuit, with a SA-1 fault $F$ on signal line $b$. The values assigned to input $a$, $b$, and $c$, constitute a (single) test for the SA-1 fault. Value 0 on line $b$ activates the fault, since it drives line $b$ to a fault-free value different from the faulty circuit value. The signals with value 1 on line $a$ and $c$ cause the fault-effect; i.e., the discrepancy between the fault-free and faulty signal value, to appear on the PO $f$, thus propagating the fault-effect to an observable output.

A sequential (logic) circuit contains feedback loops, which gives the circuit the capability to memorize information. The response of a sequential circuit depends on the current stimulus and the stimuli applied to the circuit in the past. The information from the past is usually stored in explicit memory, which is formed by one or more latches and/or FlipFlops (FFs). The Current State (CS) (or state for short) is defined as the information currently stored in memory. Sequential circuits can be further categorized as either synchronous or asynchronous, depending on whether or not the feedback loops have FFs and/or clocked latches. In a synchronous circuit the signal values are passed through the FFs in the feedback loops at discrete points in time (timeframes); i.e., only at a clock pulse new information can be stored in memory. Figure 1.2b shows a synchronous sequential circuit; each feedback loop contains a FF, which is controlled by a clock signal. As for combinational circuits, also for sequential circuits, test sets are generated by test generation algorithms. However, rather than a test vector, a test sequence is required for a single fault. Because the fault can be activated and/or propagated via (requirements on) the FFs, more than one timeframe is usually required for a test. Figure 1.3b shows an example of a sequential circuit with a SA-1 fault $F$ on line $b$. Value 0 on line $b$ in timeframe 0 activates the fault. To propagate the fault-effect on line $b$ in timeframe 0 to line $d$, value 1 is required on line $c$ in timeframe 0. Value 0 on line $e$ in timeframe $-1$ justifies this requirement on line $c$. Value 1 on line $a$ in timeframe 1 propagates the fault-effect (on line $d$ in timeframe 0) to the output (line $h$). Hence, the sequence $\{X, X, 0\}, \{X, 0, X\}, \{1, X, X\}$ on lines $\{a, b, c\}$ in timeframes $-1, 0$ and 1 is a test for fault $F$.

In this thesis, the subject of (automatic) structural test generation for synchronous sequential circuits is investigated. In the next two sections, the motivation and justification for this investigation is explained.

### 1.2 Complexity of TPG for sequential circuits

In this section the research on TPG for sequential circuits is motivated, by describing the huge (time/memory) complexity of TPG for sequential circuits. Test Pattern Generation for Sequential circuits (STPG) (as well as for Combinational circuits (CTPG)) is an NP-hard problem [Fuj82] [Fuj85]. There is no known polynomial-time algorithm that can be used to generate tests for these circuits. However, for a large class of circuits it is possible to generate tests efficiently (i.e., within reasonable time) using various heuristic search techniques. For combinational circuits, many TPG algorithms have been proposed in the past (such as [Rot66] [Goe81] [Fuj83]). Despite the exponential time-complexity (in worst case, all $2^n$ ($n = \#PIs$) possible input vectors have to be evaluated), for most
1.2. Complexity of TPG for sequential circuits

<table>
<thead>
<tr>
<th>Combinational circuits</th>
<th></th>
<th>Sequential circuits</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Wai90]</td>
<td></td>
<td>[Nie91]</td>
</tr>
<tr>
<td>Name</td>
<td>Feff(%)</td>
<td>CPU(s)</td>
</tr>
<tr>
<td>c1355</td>
<td>100</td>
<td>4.7</td>
</tr>
<tr>
<td>c2670</td>
<td>100</td>
<td>17.3</td>
</tr>
<tr>
<td>c5315</td>
<td>100</td>
<td>10.8</td>
</tr>
<tr>
<td>c6288</td>
<td>100</td>
<td>23.8</td>
</tr>
<tr>
<td>c7552</td>
<td>100</td>
<td>41.5</td>
</tr>
</tbody>
</table>

Table 1.1: Some TPG results for ISCAS combinational [Brg85] and sequential [Brg89] circuits

circuits the TPG generates a test set efficiently and reaches a high fault efficiency\(^4\). Also many STPG algorithms have been proposed (such as [Mar78] [Che88] [Sch9a] [Nie91]). However, until now the results are by far not as good as for CTPG. STPG has been recognized as a very complex problem [Lio89] [Mar95]. Table 1.1 illustrates the much higher complexity of STPG in comparison to CTPG. The first and fourth columns show the names of five combinational and five sequential circuits; these circuits are part of the ISCAS benchmarks [Brg85][Brg89]\(^5\), which are often used in research for test generation. The number in the names reflects the number of signal lines of the circuit. The columns entitled Feff give the fault efficiency of two sophisticated ATPG systems: [Wai90] for the combinational circuits and HITEC [Nie91]\(^6\) for the sequential circuits. The CPU columns give the required test generation time in seconds. Comparing the TPG results for combinational circuits with those for the sequential circuits demonstrates that TPG for sequential circuits requires much more effort, in terms of obtaining a high fault efficiency with an acceptable test generation time.

Three circuit-attributes have been identified which contribute to the increased complexity of STPG:

1. A low density of state-encoding [Mar95]; i.e., a low fraction of the total number of possible states (in memory) is functionally used (the legal states). The remaining states usually cannot be in memory; i.e., are unreachable. These states are called Illegal States (ISes). The STPG has to prove that these ISes cannot be established before requesting for another state to test the fault-under-test. This often consumes a lot of computing time. A high number of illegal states, compared to the number of legal states, is known to increase the difficulty for STPG [Mar95].

2. The huge search space. The search space per timeframe consists of the state space\(^7\) combined with the PI space\(^8\). The size of the state space is \(9^m (m = \#FF)\) and the size of the PI space is \(2^n (n = \#PI)\) [Che88]. Compared to a combinational circuit,

\(^4\)Fault coverage is defined as the percentage of tested faults from the total number of faults. Fault efficiency is defined as the percentage of tested plus proven untestable faults from the total number of faults.

\(^5\)See Appendix A for a description of these circuits.

\(^6\)These results have been published in [Rud95], which uses HITEC.

\(^7\)The space of all FF signal value combinations

\(^8\)The space of all PI signal value combinations
the search space per timeframe of a sequential circuit is much larger. In addition, a
test in a sequential circuit usually consists of a sequence of vectors. Hence, the size
of the search space exponentially increases with the number of timeframes required
to test the fault.

3. A high sequential depth and/or a high number of cycles (or feedback loops) [Lio89].
For each signal line, the sequential level is defined as the minimum number of
FFs on the paths to any PO. Figure 1.4 shows a sequential circuit with two FFs.
The sequential level of line \( a \) is 0; i.e., the path to the PO contains no FFs; while
the sequential level of line \( b \) is 1, and of line \( c \) is 2. The sequential depth of the
circuit is defined as the maximum sequential level of the signal lines. Hence, the
sequential depth of the circuit in the figure is 2. This measure gives an indication
of the minimum number of clock pulses needed to test a fault. However in [Mar95]
it has been proven that the sequential depth and the number of cycles do not make
such a large contribution to the STPG complexity as [Lio89] claimed.

The above three circuit attributes tend to reduce the “testability” of the circuit (see
Section 1.3) and, hence, increase the complexity of STPG. In addition to these attributes,
also the presence of multiple fault-effects (originating from the single (SA-)fault) during
STPG increases the complexity: The fault is present in each timeframe. Hence, a signal
line can be influenced by the fault-effect in the current timeframe and the fault-effects
of previous timeframes (via the feedback-loops). This increases the chance that fault-effects
on signal lines disappear (because two or more fault-effects mask each other), possibly
making the fault (more) difficult to detect or even not detectable. Figure 1.4 shows that
line \( d \) is influenced by the fault \( F \), on line \( b \), via the previous two timeframes: Via path
\( b, g, d \) and via path \( b, g, c, e, g, d \). A solution to reduce the STPG complexity is the
application of DFT techniques, which are described in the next section.

1.3 Design for testability and its limitations

The testability of a circuit, in the context of test generation, depends on the costs re-
quired for test generation, for test application, and on the reachable fault coverage (i.e.,
the testable faults divided by the total number of faults). The testability increases when-
ever the costs of test generation and application decrease (lower time complexity), and/or
1.3. Design for testability and its limitations

Figure 1.5: a) An example of an FF with scan capability; b) Sequential circuit with SFFs

the maximum fault coverage increases. Testability is strongly related to the controllability and observability of signal lines. The controllability refers to the estimated effort required to produce specific signal values on signal lines by applying input values on Pls. The observability refers to the estimated effort required to make signal value changes on lines visible at the POs. Hence, if the TPG effort required to control signal lines to values, and to make values on signal lines observable, in order to activate and propagate faults, is low then the circuit has a high testability.

STPG often results in requirements for the values of the FFs. These requirements have to be justified by PI assignments in previous timeframes. It is easy to imagine that extra timeframes, required to test the fault-under-test, increase the costs to activate and propagate the fault. In other words, the costs to control lines or to make lines observable increase, hence decreasing the testability of the circuit. However, the testability of sequential circuits can be improved by raising the controllability and observability of the circuit lines, using techniques known as Design For Testability (DFT). These techniques can improve the testability of sequential circuits to the level of the combinational ones. Inserting test points\(^9\) (e.g., [Wil73]) and using the Scan Design methodology [Eic77] are examples of such techniques. In the scan design approach (see Figure 1.5) all (or sometimes a fraction of) the FFs of the circuit are made scannable (i.e., their values can be directly controlled and observed); the FFs are linked together to form a shift-register, a so-called scan-chain. Figure 1.5a shows a scan-FF (SFF); in addition to the normal data-input and output, it has a scan-input, a scan-output, and a test-mode input. The test-mode input selects between the data-input and the scan-input. The scan-input is a PI (start of the chain) or is connected to the scan-output of the previous SFF in the chain (see Figure 1.5b). The scan-output of the last SFF in the chain is a PO. When the test-mode input selects the scan-input of the FFs, any signal value can be shifted into, and out of any FF. If all FFs are scannable, then the test generation effort is reduced to the effort required for a combinational circuit. Although this simplifies the task of TPG greatly, the disadvantages of this approach are penalties on circuit area, circuit performance, and test-time:

\(^9\) Test points are additional Pls and/or POs which can be used during testing to ease controlling and observing certain signal lines.
• *Scan flipflops* are required which are larger than normal FFs, hence requiring extra area.

• Because the SFFs are larger, extra delay is often introduced, degrading the performance of these FFs. This may become especially important for future generation 1GHz products.

• IC testing cannot be performed at-speed.

• Extra test-time is often required, because scanning test vectors in and out of the SFFs is necessary.

• The defect coverage is reduced. Because testing is not performed at-speed, defects that manifest themselves as delay faults are not detected. This is important for high performance designs allowing only low DPM values; which is a current trend.

In addition to the above disadvantages, often not all FFs can be made scannable: FFs in the critical path are often excluded from being scannable, due to a too large performance degradation; or FFs are not made scannable just to save area. In fact, even one or two non-scan FFs ("near full-scan") necessitates sequential circuit TPG, otherwise (some) faults will remain undetected. Nowadays often the scan design methodology is used by default, without investigating non-scan (and thus sequential) test generation. But this may incur unnecessary circuit area, circuit performance, and test-time overhead. Even though the complexity of sequential TPG is much higher than combinational TPG, the disadvantages of full-scan, or the impossibility to apply it, makes TPG for sequential (non-, partial- or near full-scan) circuits necessary.

### 1.4 Scope of this thesis

The discussion in the previous sections leads to the following conclusions:

• Structural testing is a minimal requirement for guaranteeing that the IC is free of defects, hence increasing the probability of a working IC. Therefore, ATPG is nowadays a necessary part in the design flow of an IC.

• ATPG for combinational circuits has been "solved"; more than 99% fault efficiency\textsuperscript{10} can be reached by most state-of-the-art ATPG systems within reasonable test generation time\textsuperscript{11}[vdL96].

• ATPG for sequential circuits has not been solved. The fault efficiencies for circuits beyond the very small ones are often very poor, and/or the test generation times are too long.

\textsuperscript{10}usually sufficient in industrial environments

\textsuperscript{11}Although for today's largest circuit designs, the limitations of current ATPG systems are again becoming into view.
1.5. Organization of this thesis

- The use of the scan design methodology, which reduces the test generation effort to the effort required for a combinational circuit, results in area, performance and test-time penalties, and is therefore often not (fully) applicable.

Hence, TPG for sequential circuits is nowadays a required part in an ATPG system. The research reported in this thesis focuses on ATPG for sequential circuits.

In [vdL96] the ATPG system DAT [Kon93] [vdL94b] [vdL94a] [Kon95b] [Kon95a] [Kon96b] [Kon96c] (Delft Advanced Test generation system) has been described. This state-of-the-art ATPG system, for 3-state combinational circuits using various fault models, generates (compact) test sets with very high fault-efficiencies, with low test generation times (compared to other ATPG systems). During the research reported in this thesis, this ATPG system has been modified and extended to be able to handle 3-state synchronous sequential circuits, without (no SFFs) or with partial-scan capabilities. In addition, new techniques and/or concepts have been added to reduce the (time/memory-)complexity of the STPG. Within the context of STPG, the research reported in this thesis focuses on:

- The signal value model used during STPG. We have identified shortcomings of existing signal models, which lead to incomplete and inefficient search in STPG, and pessimistic bus-conflict\textsuperscript{12} [vdL96] detection.

- Illegal State (IS) identification. Many ATPG systems have a lower performance than they could have, because they do not identify or optimize the ISes. In Section 1.2, it has been explained that the STPG often has to prove that states cannot be reached, hence are illegal. This often consumes a lot of computing time.

The target of this research is to develop an ATPG system for sequential circuits which has higher fault efficiencies and/or reduced test generation times in comparison to results published of other ATPG systems [Nie91] [Gou93] [Dar97]. Furthermore, the fault coverages/efficiencies reached for various circuits have to be correct; i.e., no pessimistic nor optimistic fault coverages/efficiencies should be reported.

Finally, the research performed in this thesis is based on the following assumptions: The circuits have a single clock which is implicitly present in the circuit and have (normally) no asynchronous reset capabilities\textsuperscript{13}. Although the proposed ATPG system can handle various fault models, we only consider the single SAF model, under logic observation of POs, and single time observation approach [Pom92a].

1.5 Organization of this thesis

The remainder of this thesis is organized as follows:

\textsuperscript{12}A bus-conflict occurs on a 3-state bus when simultaneously value(s) 1 and 0 is assigned on the inputs of the bus.

\textsuperscript{13}Instantaneous initialization of FFs to signal values 0 or 1.
Chapter 2: Circuit, signal, and fault models. To provide the background and basic terminology, in this chapter three models are discussed: The circuit model, the signal model, and the fault model. A description is given of the combinational and sequential part of the circuit model. The (primitive) elements of sequential circuits are introduced, and a theoretical basis on sequential circuits is given. To allow direct usage of the combinational circuit TPG, the iterative array model is introduced. This model “repeats” the combinational part of the sequential circuit for each timeframe during STPG; in each timeframe CTPG can be performed. A five valued signal model is introduced based on the basic logic values 0, 1 and Z. Value Z is the high impedance value to denote a floating signal line. Finally, a section is devoted to the Stuck-At Fault (SAF) model: A classification of faults is given, placing the SAF model in its context. The SAF detection condition is formalized and the set of SAFs is categorized, based on the detection condition.

Chapter 3: ATPG system structure. In order to provide additional required terminology, in this chapter the framework for an ATPG system is described. The main components of an ATPG system are identified and discussed. TPG for combinational circuits is described: The FAN algorithm, which often forms the basis in many state-of-the-art TPGs for sequential circuits, is described. Also, the characteristics of the (FAN-based) TPG in the DAT system are described; many techniques applied in this TPG can also be (re-)used in STPG, sometimes with small modifications. Finally, TPG for sequential circuits is discussed. A classification is given of existing STPG methodologies. Advantages and disadvantages of these methodologies are discussed.

Chapter 4: Problems in STPG and proposed solutions. Unsolved problems, prior art, open questions, and proposals are presented in this chapter; as far as related to the scope of this thesis. We have identified, in current (state-of-the-art) STPG systems, three problems: Incomplete and inefficient search in STPG, pessimistic bus-conflict detection, and over-specified Illegal States (ISes). These problems are related to the applied signal model during STPG and to illegal state identification. Previous work related to these problems is discussed. A study has been performed on state-of-the-art STPG methods. Useful techniques are selected from these methods to be incorporated into our STPG. From the identified problems and the previous work on STPG, twelve questions have been raised. To answer these questions three proposals have been made:

Proposal 1: Based on the combinational TPG, described in Chapter 3, a new STPG method is proposed. This system uses techniques from other STPGs and is extended with new techniques to handle sequential circuits efficiently (Chapter 5).

Proposal 2: A new signal model, the Power-Set Logic is proposed, solving the problems of incomplete and inefficient search in STPG and the pessimistic bus-conflict detection (Chapter 6).

Proposal 3: Six techniques are proposed to perform IS identification and “optimization”, solving the problem of over-specified ISes (Chapter 7).

The effectivity and efficiency of this new ATPG system, incorporating the three proposals, is demonstrated on a case study: The 80C51 micro-controller [Phi97] (Chapter 8).
Chapter 5: ATPG for sequential circuits. In this chapter, the new ATPG system for sequential circuits is described in detail (Proposal 1 of Chapter 4). This system incorporates a FAN-based STPG, and pre-processes and learning techniques to support the STPG. Pre-processes are introduced to simplify the task of the STPG part of the ATPG system. An STPG, based on the FAN-algorithm, and incorporating existing and new techniques, is described. Fault dependent cost estimates and the State Restrictor are introduced. The State Restrictor constrains the STPG by identifying Illegal States (ISes) during STPG. Three learning techniques are introduced, to lighten the STPG task: They try to find indirect implications, signal lines with a fixed value, and ISes. The utilized Fault Simulator (FS), which is a single pattern, single timeframe, single fault, explicit fault simulator, is described. This FS is also used for selecting the next target-fault during ATPG. Finally, as a strategy, additional target (S)TPG [Pom91a][Kon96b] is introduced. Parts of the material described in this chapter is planned to be submitted for publication [Kon98] [vdL98a].

Chapter 6: ATPG based on Power-Set Logic. In this chapter, the Power-Set Logic (PSL) signal model is introduced, which is a set-based signal model using the powerset (i.e., the set of all possible subsets) of the set of basic values \( \{0, 1, Z\} \) (Proposal 2 of Chapter 4). Each value in the power-set is an atomic value in the set-based signal model. The model allows, for example, to express that a signal can still become either \(0\) or an 'unknown' value of either \(0\) or \(Z\). We will show that this signal model guarantees a complete search during TPG, provides correct bus-conflict detection, and is more effective than TPG using the traditional signal models for 3-state and sequential circuits. Results demonstrate the superiority of PSL over other signal models by reaching higher fault efficiencies for industrial as well as sequential ISCAS'89 circuits, combined with a large reduction of backtracking and computing time. The material described in this chapter has been published in [vdL98b].

Chapter 7: Illegal state space identification. In this chapter, six new techniques are introduced to expand the known IS space; i.e., the space covered by the known ISes, which is stored in a list (Proposal 3 of Chapter 4). They identify over-specifications in the stored ISes. These techniques use the known ISes to generate candidate ISes, which have to be proven unjustifiable. If the candidate IS is unjustifiable, a new IS is stored in the list. The result is that the list covers a larger part of the IS space, and the number of ISes in the list is reduced. Based on the fact that prime ISes (i.e., these ISes have no over-specifications) are usually very sparsely assigned, we also present a learning process to 'explicitly' find prime ISes. To speed-up this process, we propose the use of a legal state cache, which stores successfully justified states, to avoid useless justification repetitions of these states. A data structure is presented which reduces the amount of memory required for ISes by a factor of up to ten and increases the “access-speed” to the ISes significantly. Experimental results are given for ISCAS’89 and some industrial circuits, and are compared to the results of HITEC [Nie91], DUST [Gou93] and MOSAIC [Dar97]. Parts of the material described in this chapter have been published [Kon97] and have been submitted for publication [Kon99].
Chapter 8: Case study: ATPG for the Philips 80C51. In this chapter a case study is presented to demonstrate the effectiveness and efficiency of the ATPG system, described in the previous chapters. The considered case is the Philips 80C51 micro-controller [Phi97]. ATPG has been performed on a full-scan version (i.e., all FFs are scannable), on a fully sequential version (non-scan), and on several partial-scan versions of this circuit. FFs are selected to become scannable, based on signal lines which are uninitializable or hard to control, or on signal values of lines which are unobservable or hard to observe. Results demonstrate that a reasonable number of FFs (±50%) of the 80C51 has to be scannable, to reach similar fault coverage in test generation, compared to the full-scan version. The fault coverage is reduced with ±10% percent, when only ±30% of the FFs have been selected for scan.

Chapter 9: Conclusions. This final chapter concludes with a summary of the main contributions of this thesis, and suggests ideas for future work.

Appendices A: ISCAS'85 and ISCAS'89 circuits, B: Industrial circuits, and C: DAT: Delft Advanced Test generation tool. The appendices provide details on the ISCAS benchmark circuits and the industrial circuits used throughout this thesis for experiments, as well as details on the ATPG tool DAT, in which the ideas presented in this work have been implemented.
Chapter 2

Circuit, signal, and fault models

In this chapter, terminology and assumptions on the used circuit, signal and fault models are introduced.

The circuit model, which consists of a combinational and a sequential part, used throughout this thesis, is presented. A brief overview is given of the combinational circuit model: Primitive Boolean, as well as 3-state elements are introduced. The sequential circuit model, which contains the memory elements (latches and/or flipflops) is described. Usually, sequential circuits belong to the class of Finite State Machines (FSMs). Therefore, the FSM theory is described. We assume that upon powering-up, an FSM is always in an arbitrary state. Therefore, state initialization is required to bring the circuit into a known state. Methods, terminology and related issues on state initialization are discussed in this chapter. Thereafter, the iterative array model is introduced; it "repeats" the combinational part of the sequential circuit for each timeframe. The advantage of this model is that within each timeframe test generation algorithms for combinational circuits can be applied. A five-valued signal model, used during STPG, is described. This model is based on the basic logic values \( 0, 1 \) and \( Z \). The value \( Z \) is the high impedance value to denote a floating signal line, which can occur in 3-state circuits. Finally, the SAF detection condition for sequential circuits is formalized and a classification of SAF instances is given, according to the detection condition.

This chapter is organized as follows: Section 2.1 describes the combinational circuit model. Section 2.2 describes the sequential elements (latches and flipflops) in the circuit. FSMs and state (or circuit) initialization is explained. Section 2.3 presents the iterative combinational array model of sequential circuits. Section 2.4 describes the signal model, capable of handling Boolean and 3-state elements. Section 2.5 presents the detection conditions of the SAF model, under logic observation. In addition, a classification of fault instances, according to the detection condition is given. Finally, Section 2.6 provides a summary of this chapter.
Chapter 2. Circuit, signal, and fault models

![Diagram of combinational and sequential circuit elements](image.png)

**Figure 2.1: Combinational circuit model of sequential circuit**

2.1 The combinational circuit model

In this section, the structural circuit model of the combinational logic in a sequential circuit is described. The combinational logic block (see Figure 2.1) is usually a (acyclic) gate network, defining (at a certain time instance) the mapping between the primary inputs (PIs) and the **Current State** (CS), and the primary outputs (POs) and the **Next State** (NS). The CS is the state $s_{CS}$ (or state vector $p_{pi}$) currently in the memory elements, and is present on the **Pseudo Primary Inputs** (PPIs) of the combinational circuit. The NS is the state $s_{NS}$ (or state vector $p_{po}$) which is to be stored in memory after applying a clock pulse; this state is currently on the **Pseudo Primary Outputs** (PPOs). Hence, the relation between the CS and NS is: $s_{CS}[t] = s_{NS}[t-1]$ (or $p_{pi}[t] = p_{po}[t-1]$); the CS after a clock pulse on time stamp $t$ is equal to the NS before this clock pulse. Note that a PPI is an output of a memory element; a PPO is an input of a memory element.

An **input vector** or **input stimulus**, $\vec{p}_{i}$, is defined as the data applied to the PIs at a certain time. An **output vector** or **output response**, $\vec{p}_{o}$, is defined as the data present on the POs after the circuit has become stable again, after a clock pulse. After that, the next input vector can be applied. Successively applied input vectors are called an **input sequence**, PI. Similarly, the corresponding sequence of output vectors is called an **output sequence**, PO.

The gate network is implemented using **random logic**\(^1\); i.e., Boolean and/or 3-state logic elements. The types of logic elements considered are called the **primitive elements**. The primitive elements (or **elements** for short) have zero or more input terminals and zero or more output terminals. Usually an element with multiple inputs has one output, and an element with multiple outputs has one input, as shown in Figure 2.2. The elements are interconnected using **signal lines**, also called **connections** (see Figure 2.2). Each connection has a **signal value** (sv) associated with it, which is a function of the sv's of the inputs of the connection's from-element.

A **Boolean circuit** is a circuit which is built with only Boolean elements: AND, NAND, OR, NOR, BUF (buffer), INV (inverter), XOR and NXOR. These Boolean elements are primitive elements in our circuit model. The signal values of the connections in a Boolean circuit can only be binary values: Logic 0 and 1. In addition to these values,

\(^1\)However, these functions can also be implemented using a Programmable Logic Array.
2.1. The combinational circuit model

![Diagram](image)

**Figure 2.2:** From- and to-element $E_f$ and $E_t$ from connection $c_k$ with input and output terminals

![Diagram](image)

**Figure 2.3:** a) Multiplexer using two (three-state) switches and a bus; b) three-state output pad driver

The value $X$ is defined; it denotes an unspecified sv (or don’t care sv); i.e., the sv still can become either logic 0 or 1. Hence, in Boolean logic circuits the three-valued signal model is required: $S = \{0, 1, X\}$.

Nearly all published work only considers Boolean circuits. However, current industrial circuit designs contain three-state (3-state) logic elements, such as 3-state buses, bus drivers, transmission gate logic, and three-state output pad drivers. These elements can have a third signal value, called high-impedance (commonly denoted as Z). In addition, current industrial circuits also do have bi-directional (I/O) terminals [vdL96]. Circuits consisting of Boolean as well as 3-state primitives are called 3-state circuits. Figure 2.3 shows two typical examples of 3-state constructions, frequently used in 3-state circuits: The multiplexer and the output pad driver. Three-state elements are often used to multiplex a number of signals. The signals are controlled using switches (see Figure 2.3a) whereby the switch outputs are "tied" together (using a bus); each switch is controlled so that, at most one switch is enabled (i.e., can propagate a logic 0 or 1 value to the output of the bus). In case multiple switches (driving the same bus) are allowed to be enabled, the circuit designer and/or test engineer is responsible for preventing that the bus is simultaneously driven by at least one 0 and one 1 (see Figure 2.4a). Such a situation is called a bus-conflict\(^2\). In CMOS technology, this implies that a conducting path exists from ground ($V_{ss}$) to power ($V_{dd}$), which may result in circuit damage [vdL96]. The output sv of the bus, as shown in Figure 2.4a, is unknown. This is denoted with value $U$ and occurs when an element (in this case the bus) cannot determine the output sv based on

\(^2\)There is no standard expression for this phenomenon; bus-conflict is also called bus-clash or bus-contention
the specified input values (i.e., they are not X). Hence, in 3-state circuits the five-valued signal model is required: $S = \{0, 1, Z, U, X\}$.

To model 3-state logic in circuits additional primitive elements are defined. The following 3-state primitives are considered in this thesis\(^3\).

- A **BUS** to model pure 3-state buses. Figure 2.4b shows the bus symbol and the five-valued truth table. To avoid bus-conflicts, buses have internal restrictions: Different logic values (i.e., value 0 or 1) simultaneously on the bus inputs are not allowed.

- A **SWI** and **NSWI**, to model 3-state switches as used in bus-drivers, multiplexers, transmissions gates, etc. Figure 2.5 shows the symbols for these elements. The switches SWI and NSWI propagate the value on the data input to the output when the control input has enabled (1 respectively 0) this element. If the control input disables the switch, the output value is Z.

Thus far, primitive elements, which can occur in the combinational logic of the sequential circuits have been described. The next section covers the sequential part of the circuits.

### 2.2 The sequential circuit model

In this section the sequential part of the circuit model will be described. In Subsection 2.2.1 the memory elements are described; they can be latches or flipflops. Thereafter, in Subsection 2.2.2, a class of sequential circuits is introduced: The Finite State Machine (FSM). Throughout this thesis, only these machines are considered. Properties and methods to represent FSMs are described. The last subsection introduces terminology and definitions on circuit initialization. The target of this subsection is to explain the concepts of legal and illegal states.

---

\( ^3 \)However, the 3-state circuits used in our experiments are not restricted to these 3-state elements. More primitive (3-state) elements and a more detailed description of these primitive elements can be found in [vdL96]
2.2. The sequential circuit model

![Figure 2.6: a) S-R latch; b) symbol; c) truth table](image)

**Figure 2.6:** a) S-R latch; b) symbol; c) truth table

![Figure 2.7: a) gated S-R latch; b) IEEE symbol](image)

**Figure 2.7:** a) gated S-R latch; b) IEEE symbol

2.2.1 Latches and flipflops

In order for the output of the sequential circuit to depend on past inputs, the circuit has a mechanism to retain the information of previous inputs. This mechanism is the memory of the circuit. The elements in the memory are implemented using the bistable (i.e., has two stable states: 0 and 1) elements: latches or flipflops.

**Latch.** Figure 2.6 shows the Set-Reset latch (S-R latch), together with IEEE symbol and truth table. When inputs \( S = 1 \) and \( R = 0 \) the latch is “set”; i.e., output value \( Q \) becomes 1. When \( S = 0 \) and \( R = 1 \) the latch is “reset”; i.e., output value \( Q \) becomes 0. The latch “holds” the output values when \( S = R = 0 \). The input combination \( S = R = 1 \) is not defined, because the output values then depend on which input value (both are active) is “handled” first; i.e., the output value is unpredictable⁴.

In this thesis only synchronous sequential circuits are assumed. Therefore, only gated latches are allowed. Figure 2.7 shows a gated latch; i.e., new data entered on the S-R inputs can only become active (i.e., affect the output value) when the control or clock signal (C) becomes active. Hence, the output value of the latch can only change at fixed time periods. These controlled S-R inputs are called synchronous inputs⁵. Uncontrolled inputs, such as the S-R inputs of the latch shown in Figure 2.6 are called asynchronous inputs. Figure 2.8 shows an other type of gated latch: The gated D latch. It has two advantages over the S-R latch: It has only one data input, and there is no input state which can cause unstable situations (note that by disabling the control signal C, the latch performs its “hold” function).

⁴However, sometimes precedence values are applied to the inputs; i.e., the S input can have precedence over the R input and vice versa. In that case an unpredictable output value cannot occur.

⁵Value “1” in the symbol C1 (see Figure 2.7b) represents a control dependency. The rule is that when \( C1 = 1 \), all 1S and 1R have their normal effect; when \( C1 = 0 \), the 1S and 1R have no effect.
Figure 2.8: a) gated D latch; b) IEEE symbol

Figure 2.9: a) D-flipflop implemented using 2 gated S-R latches; b) D flipflop symbol

An important characteristic of the (S-R as well as D) latches is that when an input value is changed (and when, in case of gated latches, the clock signal is active), any effect on the output appears directly after the new input value. The new output values are only delayed by the propagation times of the elements in the latch. This property is called transparency [McC86]. Transparency of the latches is the reason why often these elements are replaced by flipflops (which are not transparent): Due to the transparency property oscillation or races can occur, which can set the circuit in unpredictable states (instability) [Thi88]. Oscillation can, for example, occur when the output of a latch can directly influence the input of the same latch.

**Flipflop.** A flipflop is a gated, or clocked, bistable element that does not have the transparency property. Thus, a change of a flipflop output value is never a direct response to a change of an input value. The output value of a flipflop is called the state of the flipflop. Figure 2.9 shows a D-flipflop\(^6\). A D-flipflop is implemented using two gated S-R latches (see Figure 2.9a). Depending on clock input \(C\), either the master or slave latch is active: When \(C = 0\) the master latch is active and the \(sv\) on the \(D\) input determines the output \(sv\) \(Y\) of the master latch. Note that the output value \(Q\) does not change, because the slave latch is inactive. When \(C\) changes to \(1\), the slave latch becomes active and the output value of the master latch is copied to the output value of the slave latch. Hence, to propagate a \(sv\) on input \(D\) to output \(Q\), \(C\) has to become successively \(0\) (to "load" the \(sv\) on \(D\) in the master latch) and \(1\) (to propagate the \(sv\) from the master- to the slave latch). This is the main difference in operation compared to the D-latch; where as long as the clock input \(C\) is \(1\), every change in data value on the input is "immediately" propagated to the output.

Although a flipflop is a more complex and costly element than a latch, it is often

---

\(^6\)Other flipflop types, such as the J-K flipflop and T-flipflop are not considered in this thesis. For more information see [McC86] or [Thi88].
preferred because instability in a circuit with flipflops can hardly occur. This simplifies circuit design. The reason is, that the state of a flipflop can only change on the positive edge (i.e., an $0 \rightarrow 1$ or up transition) of the clock input $C$. Hence data input value changes (even if caused by the FF itself) have no effect on the flipflop as long as the clock input $C$ remains $0$ or $1$; only $C$ input changes from $0$ to $1$ can result in state changes. Such flipflops are called positive edge triggered flipflops\textsuperscript{7}.

Asynchronous inputs. Some synchronous sequential circuits allow flipflops to have asynchronous inputs; i.e., inputs which directly affect the state of the flipflop. Two examples are the reset input and the set input. These inputs reset or set the flipflop state. Figure 2.10 shows the symbols. The (re)set input is often used to place the flipflop in a fixed initial state when starting up its operation, and sometimes also to control its state during testing\textsuperscript{8}.

2.2.2 Finite state machines

A special class of synchronous sequential circuits is the Finite State Machine (FSM). Such a machine can be described as a 5-tuple:

$$M = (PI, S, PO, \delta, \lambda)$$

where $PI$ is a finite, non-empty set of primary inputs; $S$ is a finite, non-empty set of (circuit) states; $PO$ is a finite, non-empty set of primary outputs; $\delta : S \times PI \rightarrow S$ is the next-state function, and $\lambda : S \times PI \rightarrow PO$ is the output function\textsuperscript{9} (The functions are implemented by the combinational logic block (see Section 2.1)). An FSM based on these functions is called a Mealy machine [Mea55]. An FSM whose outputs depend only on the flipflop outputs ($\lambda : S \rightarrow PO$) is called a Moore machine [Moo56]; i.e., a Moore machine is a special case of the Mealy machine [Thi85].

\textsuperscript{7}Of course, also flipflops exist which trigger on the negative clock edge (state changes on down transition of the clock), on both edges, or trigger on level changes of the clock. Even flipflops exist which trigger on multiple clocks [McC86].

\textsuperscript{8}Two reasons exist for implementing a separate (re)set input on flipflops (and hence, requiring extra separate reset circuitry): Circuit reset is such a common function resulting in a standard design for reset circuitry using asynchronous inputs of the flipflops. The second reason is that the reset function can be added without modifying the circuitry, designed for the specific circuit function (however, the physical design is influenced, because extra wiring is required).

\textsuperscript{9}Sometimes an FSM is described as a 6-tuple; the sixth item is $s_0$, being the set of initial states. If the circuit can in any state after power-up, $s_0$ is equal to $S$.}
Figure 2.11: a) State transition graph; b) State transition table

In addition to the properties of a sequential circuit (e.g., the output values at any time depend on the present and past input values), an FSM has the following properties [McC86]:

- **Single-clock property.** There is only one clock input. Output changes occur only in synchronism with this clock-input.

- **Single edge property.** All internal state changes occur in response to the clock input changing from 0 to 1. A change of the clock input from 1 to 0 never causes an internal state change.

- **External-trigger property.** An internal state transition never causes a subsequent internal state transition.

The circuits considered in this thesis belong to the class of FSMs. Because of the properties of FSMs, the clock can be implicitly present in the circuit; i.e., in the circuit model no clocks pins are present.

The actions of an FSM can be described using a *state transition table* (STT), or in graphical form using a *state transition graph* (STG) or *state transition diagram* (STD). The STG describes the next-state function (δ) of a sequential circuit. Figure 2.11a shows an STG of an FSM with 2 flipflops (hence it has four states). The arrows in the graph denote state transitions. The labels of the arrows are the input (/../) and output (/../) vector of the circuit; the input vector is the condition required for the state transition. For example, a state transition from state S1 to state S2 occurs when input value PI = 1 is applied, resulting in output value PO = 1. The STG of the figure is a Mealy machine because each state and input combination has its corresponding output. In a Moore machine, the output PO would be specified in the nodes of the graph, because each state has its output PO independent of input PI. Figure 2.11b shows the STT corresponding to the STG of Figure 2.11a. The STT shows for each input PI and current state CS, the output PO and the next state NS. The STT contains exactly the same information as the STG. The STT is usually easier to use for optimizing/minimizing the next-state function, while the

\[\text{CS} = \text{Current State} \quad \text{NS} = \text{Next State}\]
2.2. The sequential circuit model

Figure 2.12: a) + b) Both circuits are uninitializable, because there exist no input sequence which bring the circuit into a known and unique state

STG may be more convenient for visualizing the circuit operation. Both STG and STT are only useful for small circuit designs.

2.2.3 Circuit state initialization

Upon power-up, an FSM is always in an arbitrary state. There are two methods to drive the circuit into a known state: Apply an asynchronous reset (see Section 2.2.1, Page 19) or apply an initialization input sequence. Both methods bring the circuit into a known and unique (specific) state. An asynchronous reset is a quick and easy way to bring (some of the) flipflops in a known state (usually called the reset state). Disadvantages of an asynchronous reset are the extra wiring and somewhat more costly flipflops. Except when explicitly mentioned, during the remainder of this thesis all FSMs are assumed to have no asynchronous reset, because in that case the worst case STPG scenario is assumed.

An initialization or synchronizing sequence is defined as an input sequence that brings the circuit to a known and unique state, that can be determined without any knowledge of the output response or the initial state [Pom94a]

Thus, a circuit is uninitializable, when no input sequence exists which brings the circuit into a known and unique (specific) state. Figure 2.12a shows an STG of an FSM which is uninitializable. After power-up the initial state is unknown; the initial state can be either S1, S2, S3, or S4. There is no input sequence which brings the circuit from the initial (power-up) state into a known and unique state (i.e., either S1, S2, S3, or S4). An asynchronous reset is required to bring the circuit into a known, specific state. After the (asynchronous) reset, either state S1, S4, or \{S2, S3\} is reachable; i.e., only one or two states are used\(^{11}\). Hence, to have an initializable circuit, the corresponding STG has at least to be connected; i.e., for every pair of states Si and Sj there is always a path from Si to Sj or from Sj to Si. Later in this section we show that this is a necessary, but not a sufficient condition for being initializable. For example, Figure 2.12b shows the STG of a circuit. The STG is connected; however, the circuit is not initializable: Every possible input sequence of arbitrary length, results in a final state which is either S1 or S2, but it is unknown which of those two.

\(^{11}\)Note: In this case, this FSM could also be implemented using one flipflop, making it (probably) initializable.
Figure 2.13: a) An FSM with legal and illegal states; b) State space of an FSM; Legal states are reachable by all states

Figure 2.14: Traffic-lights: a) Legal states and transitions; b) Illegal state

Figure 2.13a shows a connected STG of an FSM. States S1, S2, S3 are reachable from all other states; these states are called legal or valid states. State S4 can not be reached from the other states (not necessarily all states); it is defined as an illegal or invalid state.

A legal state is defined as a state which is reachable from all other states after applying a specific input sequence; i.e., starting from an arbitrary state, an input sequence exists, which brings the circuit into the specified legal state.

An illegal state is defined as a state which is unreachable from one or more states; no input sequence exists which brings the circuit to this state, starting from the unknown initial state; the unknown initial state covers the complete set of states.

Thus, the STG has to be at least connected to have legal states. Figure 2.13b visualizes the illegal and legal state concept: An input sequence exists to bring the circuit from an illegal state to a legal state; but there is no way back.

The concept of legal and illegal states can be illustrated with the following example. Figure 2.14 shows a traffic-light. The states red, green and yellow are legal states; they are reachable from all power-up/initial states. Also the state transitions from red, to green and then yellow is allowed (see Figure 2.14a); the order of state transitions is defined in the STG. However, there is no state with simultaneous red and green light (Figure 2.14b). Such state is unreachable (unjustifiable) and thus illegal.

The legal states, which form a subgraph, are always strongly connected [Che90a]; i.e., for every pair of (legal) states Si and Sj, there is a directed path from Si to Sj as well as one from Sj to Si.
2.3. **Iterative array model of sequential circuits**

![Diagram showing states S1, S2, S3, S4 with transitions labeled with 0 and 1. All states are illegal.](image)

**Figure 2.15:** An strongly connected STG is not a sufficient condition for legal states

**Proof:** S1 and S2 are legal states. Assume, there is no path from S1 to S2. But a legal state is reachable from all other states, so this contradicts. Assume, there is no path from S2 to S1. But a legal state is reachable from all other states, so this contradicts. Hence, S1 and S2 are strongly connected.

**Note:** A strongly connected STG does not imply that the states are legal. Figure 2.15 shows an example of a strongly connected STG. However, all states are illegal, because no input sequence exists to bring the circuit in either state S1, S2, S3 or S4, starting from an arbitrary state, using one single sequence.

* A circuit with legal states always has an initialization sequence.

**Proof:** This follows from the definition of a legal state.

All legal states of a circuit form one (terminal) strongly connected component of the STG. Figure 2.13a and b show the terminal property of the legal states in the STG; i.e., legal states are reachable by all other states (including the illegal states) and illegal states cannot be reached by legal states.

The circuits used in this thesis usually require an initialization sequence in order to be testable; the FFs have no (re)set property. This means that the STG of these circuits have to have a terminal strongly connected component with the legal states. As described in Section 1.2, this component is usually a very small part of the total STG: Most states are illegal. Faults which require these ISes in order to be detected, are untestable; i.e., if during TPG an IS is required for a fault, the TPG can start backtracking. Therefore, knowledge of these ISes is very important for successful test generation (see Chapter 7); wasting time in (repeatedly) justifying ISes is then avoided.

The requirement that a circuit has to be completely initializable with a synchronizing sequence is often too severe. In many industrial circuits nowadays, a (large) fraction of the FFs in the circuit have an asynchronous (re)set or are scannable, to make initialization of the circuit possible. E.g., in Chapter 8 TPG is performed on the Philips 80C51 circuit, which is only initializable with FFs with an asynchronous (re)set (and scannable FFs, during testing).
2.3 Iterative array model of sequential circuits

This section describes the iterative array model, as often used in STPG, in order to make usage of combinational TPG techniques possible. The iterative array model (see Figure 2.16) consists of an array of the combinational part of the FSM at fixed points in time. Each combinational part \( C \) represents a timeframe. Two timeframes are connected with the memory block \( M \), formed by gated latches or flipflops. Hence, the iterative array model 'rolls out' the FSM in time. After each clock pulse a new timeframe is required.

A timeframe (see Figure 2.16) at time stamp \( t \) has input vector \( \vec{v}[t] \), output vector \( \vec{r}[t] \), current state \( \vec{p}[t] \), and next state \( \vec{p}[t+1] \). Current state \( \vec{p}[t+1] \), is equal to the next state \( \vec{p}[t+1] \); also \( \vec{p}[t-1] = \vec{p}[t] \). \( s[0] = \vec{p}[0] \) is the power-up or initial state, which is an unknown state. Starting with initial state \( s[0] \), the input sequence \( \Pi = \{ \vec{r}[0], \vec{r}[1], \ldots, \vec{r}[k] \} \), applied at the PIs at timeframes \( t = 0, 1, \ldots, k \), results in an output sequence \( \Omega = \{ \vec{p}[0], \vec{p}[1], \ldots, \vec{p}[k] \} \), captured from the POs, and state sequence \( \mathcal{N} = \{ \vec{p}[0], \vec{p}[1], \ldots, \vec{p}[k] \} \). The FSM is thereby transformed into a combinational circuit. Each clocked memory element in \( M \) is modeled as a PPI with corresponding PPO; this way the memory elements behave as simple buffers (often referred as pseudo flipflops [Gho92] [Fuj85]).

2.4 The 5-valued signal model for 3-state circuits

This section describes the 5-valued (5UV)\(^{12}\) signal model, which is a basic signal model for 3-state circuits. However, we show that this signal model lacks resolution\(^{13}\), resulting in a pessimistic view concerning bus-conflicts. Therefore, in [vdL96] the so-called bus-
2.4. The 5-valued signal model for 3-state circuits

Figure 2.17: a) A bus-conflict; Bus-patch solves incorrect bus-conflict detection (b) and improves the resolution of the 25V signal model (c)

Patch procedure has been introduced, which will be described in the second half of this section.

In addition to the Boolean logic values 0 and 1, in Section 2.1, values Z and U have been introduced to be part of the signal model. Value Z denotes a high impedance value. E.g., Z is the output of a switch which is disabled. Value U has three purposes in 3-state sequential circuits:

1. To denote an output value of an element which can not be further specified (we say that this value is fixed.). However, the output value is unknown. For example, the bus, shown in Figure 2.17a, has output value U.

2. To denote the value of a PI which is not allowed to be assigned a value. This is imposed by the fixed vector which is part of a Restrictor [vdL96]. Sometimes PIs are not allowed to be controlled by the TPG because the PIs can only be controlled by other modules and not by the user or the ATE.

3. To denote the value of a FF (PPI), which is not allowed (anymore) to be assigned value 0 or 1. E.g., the PPIs of the initial state of the circuit are assigned value U, to form the unknown power-up state.

Note that the value U is a specified and unknown value, and the value X is an unspecified and unknown value. Thus all functions of the primitive elements described in Section 2.1 are a map \( f : S^n \rightarrow S \), wherein \( S = \{0, 1, Z, U, X\} \), and \( n \) is the number of element inputs.

For bus elements a problem arises when the 5UV signal model is used. Consider the situation in Figure 2.17b, the bus output evaluates to U, and a bus-conflict is detected. However, because the switches in Figure 2.17b have the same data value (0), the output value can only become 0 or Z, and therefore, a bus-conflict can never occur. Hence, the 5UV signal model has a pessimistic view concerning bus-conflicts. Therefore, the so-called bus-patch [vdL94b] [Kon95b] was introduced. It analyzes all X and U values on outputs of 3-state primitives for their 'real' meaning. Figure 2.17b and c show two examples to demonstrate the bus-patch. The bus-patch correctly evaluates the bus output value and reports no bus-conflict (Figure 2.17b). The output of the bus, shown in Figure 2.17c,
evaluates to $X$ instead of $0$ because the 5UV signal model cannot express the fact that the output of the lower switch can only become $0$ or $Z$. The bus-patch solves this problem: It analyzes the outputs of the bus and switches, and correctly evaluates signal value of the bus output to $0$.

2.5 The stuck-at fault model

In this section, the (Single) Stuck-At Fault (SAF) model is introduced, as used in this thesis. It has been introduced by R. Eldred [Eld59] and (still) is the most widely used fault model, because it is easy to apply in test generation algorithms, it is easy to imagine and understand, and the faults can be enumerated (the number of SSAFs in a circuit is two times the number of signal lines). However, many test researchers are sceptical about the quality of the SAF model, because it would cover only a small part of the possible defects [Sod90][Sto91]. Therefore, often additional tests are generated using other fault models to increase the defect coverage[Haw94]. In Subsection 2.5.1, a classification of fault models is given to place the SAF model in its context. In Subsection 2.5.2, the (SA-)fault detection condition is formalized for sequential circuits. Finally, in Subsection 2.5.3, fault instances (SAFs) are classified according to the detection condition.

2.5.1 Classification of fault models

Faults can be classified as follows:

- **Logical faults.** They cause the logic function of the circuit to be changed to some other logic function. Many fault models have been developed within this class, such as the SAF model. The SAF model assumes that each signal line in a circuit can become stuck at a fixed logic value; i.e., stuck-at-0 (SA-0) and stuck-at-1 (SA-1). Figure 2.18a shows a SA-0 fault on signal line $b$ and a SA-1 on line $a$; the signal value on line $a$ is $0$ in the fault-free circuit and $1$ in the faulty circuit. An another example: A bridging fault (see Figure 2.18a) models a permanent short between two circuit lines; e.g., the bridge between line $c$ and $d$ in the figure.
2.5. The stuck-at fault model

- **Parametric faults.** These faults alter one or more circuit parameters, such as the current consumption or the circuit speed. **Iddq faults** model changes in current consumption. Figure 2.18b shows a CMOS inverter with a defective n-transistor: This defect causes an increased Iddq\(^{14}\). **Delay faults** model defects which result in extra circuit delay, due to slow gates. A **gate delay fault** assumes a too large delay (\(\delta \) in Figure 2.18a) on a single gate; a **path delay fault** assumes a too large delay on a circuit path, which is a path between two points in the circuit, usually a PI and a PO.

Although, we only consider the SAF model, a large part of the techniques and algorithms described in this thesis can also be applied to other fault models (sometimes with minor changes).

### 2.5.2 Stuck-at fault detection condition

The detection condition of SAFs consists of the *activation*, the *observation*, and the *conflict-free condition*. In [vdL96], these conditions have been reported for combinational circuits. The fault SA-v (\(v \in \{0,1\}\)) on signal line \(c_i\) is detected iff:

1. The fault is *activated* (the activation condition): The signal line \(c_i\) with the SA-v is driven to a signal value opposite to the stuck-at level\(^{15}\):

\[
SAF_A(c_i, v) \iff (sv(c_i) = f_{c_i}(\bar{p}) = \bar{v})
\]  
(2.1)

2. The fault-effect is *observed* on at least one PO (the observation condition): The fault-effect is propagated to one or more POs, causing one or more Boolean discrepancies on these POs; i.e., one or more POs have opposite response values for the good (fault-free) and the faulty circuit.

Given that for a PO \(PO_j\), \(sv(PO_j)\) is a function of the Pls; i.e., \(f_{po_j}(\bar{p})\), and that \(sv(c_i)\) is also a function of the Pls; i.e., \(f_{c_i}(\bar{p})\), then \(sv(PO_j)\) can also be described as a function of \(sv(c_i)\) and the signal values at Pls:

\[
sv(PO_j) = f'_{po_j}(sv(c_i), \bar{p})
\]  
(2.2)

In \(f'_{po_j}\) a fault-effect at \(c_i\) can be ‘injected’, by replacing \(sv(c_i)\) by the SAF level. Logic observation at PO \(j\) of a stuck-at \(v\) at signal line \(c_i\), can thus be formulated as:

\[
SAF_O^{PO_j}(c_i, v) \iff (f'_{po_j}(v, \bar{p}) = f_{po_j}(\bar{p}))
\]  
(2.3)

Logic observation at one or more POs can thus be formulated as (\(k = \#PO\)):

\[
SAF_O^{PO}(c_i, v) \iff \left( \sum_{j=1}^{k} SAF_O^{PO_j}(c_i, v) \right) \geq 1
\]  
(2.4)

\(^{14}\)The Iddq is defined as the current in a CMOS circuit in rest; i.e., the values of the signal lines do not change. The current consumption of a CMOS circuit in rest is (very) low; an increased Iddq often is an indication of a defective circuit.

\(^{15}\)Sometimes the fault can be activated with value Z. This is only allowed when the circuit “influenced” by the fault is capable to transform the Z value to a Boolean value [vdL96].
3. All internal restrictions and external restrictions are obeyed (the conflict-free condition); i.e., no bus-conflicts in the fault-free circuit and the test pattern satisfies the restricter [Kon93].

The difference between the fault detection condition in a sequential and a combinational circuit is time: In a combinational circuit, the time instances of activation \(t_{act}\) and observation \(t_{obs}\) are equal: \(t_{act} = t_{obs}\). In a sequential circuit, the time instances of activation and observation can be different and are “somewhere” during the application of the \(p\) input vectors of the test pattern:

\[
t_{act} \leq (t_{obs} = p); t_{act} \in 0 \ldots p
\]

(2.5)

Hence, a test pattern in a combinational circuit is a single input vector, \(p\), with corresponding output vector, \(p\bar{v}\), which activates the fault and makes the fault observable at one or more POs; a test pattern in a sequential circuit is an input sequence \(\{p[0], p[1], \ldots, p[t_{act}], \ldots, p[t_{obs} = p]\}\) with corresponding output sequence \(\{p\bar{v}[0], p\bar{v}[1], \ldots, p\bar{v}[t_{act}], \ldots, p\bar{v}[t_{obs}]\}\) (test sequence), which activates the fault at time instance \(t_{act}\) and observable at time instance \(t_{obs}\).

Hence, Equations 2.1 and 2.4 have to be transformed to the time domain. The SAF detection condition for a sequential circuit is: The fault SA-\(v\) (\(v \in \{0, 1\}\)) on signal line \(c_t\) is detected iff:

1. The fault is activated on \(t_{act}\):

\[
\exists t = t_{act}, SAF_{A}(c_t, v) \leftrightarrow (sv_t(c_t) = f_{c_t}(p\bar{v}[t], p\bar{p}[t]) = \bar{v}; t \geq 0)
\]

When \(t_{act} > 0\) then a justification sequence has to be generated, which is an input sequence which traverses the CS of the circuit from the initial state \(CS = s[0]\) to the state \(CS = p\bar{p}[t_{act}]\); \(s[0]\) is the all-unknown state; i.e., \(s[0] \subset S\), which is the set of possible states.

2. The fault-effect is observed on at least one PO on \(t_{obs}\):

\[
\exists t = t_{obs}, SAF_{O}(c_t, v) \land t \geq t_{act}
\]

When \(t_{act} < t_{obs}\) then logic observation is required on one or more PPOs on time stamps \(t_{act} \leq t < t_{obs}\). Between \(t_{act}\) and \(t_{obs}\) a differentiating sequence has to be generated, which is, starting from the fault-free state \(p\bar{p}o[t_{act}]\) (\(= NS[t_{act}]\)) and the (different) faulty state \(p\bar{p}o'[t_{act}]\), a sequence of inputs vectors, such that:

\[
\forall t_{act}^{-1} SAF_{O}(c_t, v)
\]

is true.

3. All internal and external restrictions have to be obeyed for each timeframe.
2.5. The stuck-at fault model

![Fault universe divided into different fault sets, based on detectability (fault set sizes are arbitrary)]

**Figure 2.19:** Fault universe divided into different fault sets, based on detectability (fault set sizes are arbitrary)

![STG of good circuit; b) STG of faulty circuit. Fault is potentially detectable](image)

**Figure 2.20:** a) STG of good circuit; b) STG of faulty circuit. Fault is potentially detectable

### 2.5.3 Classification of fault instances according to the detectability condition

Basically, an ATPG system generates a test set, thereby dividing the fault “universe” into three sets of faults (see Figure 2.19):

A. The set of *detected faults*: The faults for which the ATPG has successfully generated a test pattern; i.e., the detection condition is satisfied (see previous subsection). A fault is called *combinationally detectable* when the fault is detected with a test sequence $T$ of length 1. The set of combinationally detectable faults is a subset of the set of detectable faults.

B. The set of *untestable or undetectable* faults: The faults for which the ATPG has proven that no test exists for that fault.

C. The set of *undetected (aborted) faults*: The faults for which the ATPG has failed to generate a test pattern, or failed to prove untestability due to time limitations.

The set of untestable faults can be further divided into two categories:

1. *Potentially detectable* [Che91] or *probabilistically detectable* [Pom93a] faults. A potentially (or probabilistically) detectable fault cannot be “hard” detected; i.e., the detection condition defined in the previous section cannot be satisfied. However,
this fault can be potential detected; i.e., after applying a pattern, one or more outputs have a known value (0 or 1) in the good case, and an unknown value in the faulty case.

There are two reasons which cause these faults to be untestable. Firstly, in 3-state circuits the faults cause an unknown value at one or more POs (e.g., the faults cause permanently closed switches, resulting in output value Z, or bus-conflicts resulting in output value U on buses; both values are propagated to POs). Secondly, they cause that legal states have become illegal in the faulty circuit; i.e., due to the fault the circuit has become uninitialized (the faulty circuit has no synchronizing sequence). Figure 2.20a shows the STG of a (good) circuit. States S0, S1 and S2 are legal states and S3 is an IS. Figure 2.20b shows the STG of the faulty circuit; the fault causes that legal state S0 becomes illegal (dotted line). This fault is (only) potentially detectable.

For both reasons a test sequence T exists that produces an output sequence which differs between the good and the faulty circuit; however, the output sequence of the faulty circuit is unknown; i.e., there are POs with value 0/U or 1/U (or 0/Z or 1/Z). Because the initial state of the circuit changes each time after power-up, repeated application of test sequence T increases the probability that the fault is detected. An ATE can not report value U or Z in the faulty case, but usually it senses these values as value 0 or 1, depending on the initial (power-up) state of the circuit and the order of the applied tests (memory retention of the 3-state elements).

2. Sequential redundant faults (SRF) [Gho92]. A SRF is a fault which cannot be activated or whose effect cannot be observed on a PO, using any input sequence, starting from the initial vector s0 (which can be any initial vector); i.e., the fault does not change the behavior of the circuit. A subset of the SRF are the Combinatorially Redundant Faults (CRFs); the fault-effect of these faults cannot be observed to any PO or PPO, beginning from any state and using any input vector. SRFs can be classified into three categories [Dev90]:

(a) Invalid-SRF: The fault requires ISes (in the fault-free circuit) to be activated or propagated; i.e., the fault did not corrupt any edge between legal states of the STG.

(b) Equivalent-SRF: The fault causes the interchange/creation of equivalent states in the STG. Hence, no differentiating sequence can be generated.

(c) Isomorph-SRF: The fault results in a faulty circuit that is isomorphic to the fault-free circuit but with a different state encoding.

Consider the circuit whose STG is shown in Figure 2.20a. Faults modify this circuit to another one having one of the STGs shown in Figure 2.21. The corrupted edge is shown using a dotted line.

Figure 2.21a is an example of a fault of the category invalid-SRF. Only the edge of the IS (S3) has been changed. This fault can never be detected because this state is unreachable in the good circuit.
Figure 2.21: a) Invalid-SRF; b) Equivalent-SRF; c) Isomorph-SRF

Figure 2.21b is an example of a fault of the category equivalent-SRF. States $S1$ and $S2$ (see Figure 2.20a) are equivalent states; they have the same behavior in the good circuit. Therefore, the corrupted edge shown in Figure 2.21b does not alter the function of the circuit.

Figure 2.21c is an example of a fault of the category isomorph-SRF. The state codes of $S1$, $S2$ and $S3$ are interchanged. The first two categories of SRFs are very common in practical circuits, while the third one is very rare [Gho92]. Therefore, many untestable faults can be identified when the ISes of the circuits are identified (see [Lon95], and Chapter 7).

2.6 Summary and conclusions

To provide background and terminology in the field of sequential circuits and test generation, in this chapter, the synchronous sequential circuit model, the 5UV signal model and the SAF fault model, used throughout this thesis, have been presented. The combinational circuit model has been defined; the primitive Boolean and 3-state gate-level elements have been introduced. The memory elements, latches and flipflops, have been described. Latches have the property that they are transparent; FFs are not transparent. This property of FFs is the reason that these memory elements are often preferred. The class of finite state machines (FSMs) has been presented. FSMs have the single clock and the single edge property. To visualize and present FSMs, state transition graphs (STGs) and state transition tables (STTs) have been introduced. Using STGs and STTs, definitions and theorems on state initialization, legal and illegal states, together with their relationships, have been presented. The circuits must have an initialization sequence and/or asynchronous (re)set capabilities to initialize the FFs. States which are unreachable from the initial state of the circuit have been defined as ISes. The iterative array model has been introduced. This model makes the usage of combinational test generation algorithms for sequential circuits possible. A 5UV signal model has been defined for sequential Boolean and 3-state circuits. The SAF detection criterion for sequential circuits has been defined. And finally, a classification of the fault universe, based on the detectability of the fault, has been given.
Chapter 3

ATPG system structure

To provide proper background and additional terminology, the framework for an ATPG system is described; it employs the circuit, signal model, and fault model, as introduced in Chapter 2. In the first part of this chapter, the main components of an ATPG system are identified and discussed. These components range from pre-processes to guide the TPG process, fault-independent (simulation-based) and fault-oriented (deterministic) test pattern generators, fault simulators, learning processes to lighten the task of the TPG process, to post-processes, such as test set compaction.

The second part of this chapter studies one main component of the ATPG system: Deterministic TPG (DTPG). DTPG methods for combinational circuits are described: The FAN algorithm [Fuj83], which nowadays usually forms the basis for state-of-the-art sequential circuit DTPGs (STPGs), as well as the main characteristics of the (FAN-based) TPG in the ATPG system DAT [vdL96] are described. Our STPG method, proposed in Chapter 4 and described in Chapter 5, is based on the FAN algorithm and uses various techniques and methods introduced in DAT. Finally, DTPGs for sequential circuits are described: Existing STPG methodologies are classified and evaluated; advantages and disadvantages of these methodologies are discussed.

This chapter is organized as follows: Section 3.1 presents the main components of an ATPG system. Section 3.2 describes DTPGs for combinational circuits. Section 3.3 describes DTPGs for sequential circuits, by classifying and evaluating existing STPG methodologies. Section 3.4 concludes this chapter with a summary.
3.1 Main components of an ATPG system

In this section the main components of an ATPG system are described; see Figure 3.1. An ATPG system requires the following inputs: A (gate-level) circuit description, a fault list, and optionally extra input information to support the ATPG process (such as Restrictors [Kon93]). Based on these inputs, an ATPG system generates a test set. In addition, the ATPG system delivers a list of detected, untestable, and undetected faults.

The six (main) components, which are shown in boldface in the figure, are explained in the next six paragraphs.

1) Pre-processes. Pre-processes are performed before test generation, and mainly aim at guiding the TPG process. For example, (sequential) controllability and (sequential) observability costs are calculated. Almost all ATPG systems use controllability and observability analysis based on, or similar to, SCOAP [Gol79].

Another pre-process is fault collapsing, which reduces the number of targets for TPG in the fault list: One fault is selected from each group of equivalent faults; the remaining faults are removed from the fault list.

A third pre-process is the identification of untestable faults. Certain faults in the circuit can be marked untestable immediately, by inspection. For example, the SA-0 fault at the enable-input of the switch in Figure 3.2 is untestable, because it cannot be made

---

\[1\]Two faults are equivalent, with respect to the considered test method(s), when no test exists which distinguishes between these two.

\[2\]however, the fault might be detectable under the potential or possible detection approach (see Sec-
3.1. Main components of an ATPG system

Figure 3.2: Switch can not propagate fault through enable input

Figure 3.3: DTPG example for SA-0 fault on line c

observable at the output; the fault-free value 1 enables the switch, and thus propagates the data value d to the output; the faulty value 0 disables the switch resulting in the unobservable high impedance value Z on the output.

2) Deterministic test (pattern) generation. In the case of an exhaustive test, for a circuit having n PIs, $2^n$ test vectors have to be applied. It is infeasible to apply them all; therefore Deterministic (fault-oriented) Test Pattern Generation (DTPG) algorithms have been developed to select the "best" test vectors [Rot66]; i.e., a minimal number of vectors which detects most faults. A DTPG generates test patterns for selected faults. Basically, two requirements have to be fulfilled during test generation for a (SA-)fault: The fault has to be activated, and a sensitive path has to be set up from the fault to a PO (i.e., the fault-effect has to be propagated to a PO; fault propagation). Both requirements result in required signal values on signal lines. These signal values have to be justified (line-justification problems); i.e., all signal values inside the circuit ultimately must be supported by the PI values. The TPG tries to solve these line-justification problems in a process which alternately implies mandatory assignments and makes heuristic choices/decisions on signal lines.

Mandatory assignments are all assignments in the circuit which can be implied due to (local) reasoning and are required to test the fault; this process is called implication. E.g., to activate the SA-0 fault in Figure 3.3, value 1 on line c is a mandatory assignment. Mandatory assignments can be identified by various techniques (e.g., fault activation, dominators [Kir87] [Sch88b], recursive learning [Kun92]). In addition, after each heuristic decision, new mandatory assignments may be found
Heuristic choices are required in the TPG process because, in order to solve one (sub)problem, usually multiple solutions exist. This is easy to imagine using the example in Figure 3.3: Signal value 1 on line c, the output of a 2-input NAND gate can be justified by applying signal value 0 on input line a or on input line b; the TPG process has to make a decision. All decisions are stored on a stack, because in general, simultaneously solving the various line justification problems may lead to conflicting assignments. In that case, the process has to fall back to the last made decision on a signal line, and either remakes this decision (i.e., make an alternative choice on that signal line) or discard this decision, if no alternative choices exists anymore, and continue to the next last decision. This process is called backtracking.

The main cause of conflicting assignments in the circuit is reconvergent fanout, which is the fundamental difficulty in test generation; due to reconvergent fanout, the line-justification problems are mutually dependent. In Boolean circuits without reconvergent fanout, no conflicts can occur. In Figure 3.3, the decision on line a resulted in activation of the fault; however, via the fanout, in a blocked element e; i.e., the fault can not be propagated anymore via line e.

The DTPG process as described so far is the basis for most DTPG algorithms and consists of traditional "path-searching" algorithms using branch and bound techniques to enumerate (implicitly) the search space. Hence, these algorithms in principle can guarantee complete test generation: If a test exists for a fault, it will be found; if no test is found, this implies that the fault is untestable. There are three DTPG algorithms for combinational circuits which often form the basis for other DTPGs for combinational as well as sequential circuits. These basic algorithms are the D-algorithm [Rot66], the PODEM algorithm [Goe81] and the FAN algorithm [Fuj83].

The first known sequential circuit TPGs (STPGs) were based on the D-algorithm: The extended D-algorithm [Kub68],[Put71], EBT [Mar78] and STG1 [Mal85]. Whereas, HITEC [Nie91] and FASTEST [Kel89] are examples of STPGs based on the PODEM algorithm. Nowadays, most state-of-the-art STPGs are based on the FAN-algorithm: Essential [Aut91], DUST [Gou93], and MOSAIC [Dar97].

The ATPG process usually consists of more than one ATPG stage, whereby consecutive stages apply increasingly powerful TPG methods, which are also increasingly expensive in terms of computing time. An ATPG stage (see Figure 3.1) consists of one or more TPGs combined with a fault simulator (see next paragraph). The first ATPG stage(s) are usually relatively "weak" and computationally inexpensive stages; in that way, many relatively easily testable faults and/or easily provable untestable faults are quickly removed from the fault list. It is a well-known fact that one specific TPG often is not successful for all faults [Kon96c]. Therefore, nowadays state-of-the-art ATPG systems incorporate multiple TPGs within ATPG stages, each using different strategies to generate a test for the faults [His96] [vdL96] [Dar97]. For example, a well known strategy in DTPG is reversing the order of search over the inputs of gates, to select the "easiest" input. A gate may have multiple "easiest" inputs, as is illustrated in Figure 3.4: The values 5, 5, and 8 on lines a, b, and c represent the controllability cost estimate for value 0 on those lines. Line a and b have both the lowest cost value; i.e., depending on the search order over the
3.1. Main components of an ATPG system

![Diagram of a NAND gate with search order and line selection](image)

**Figure 3.4:** Depending on the search order over the inputs of the NAND gate, line a or line b (lowest cost value) is selected to justify the gate.

inputs of the NAND gate (from a to c, or from c to a) line a or line b is selected to justify the gate.

3) Fault simulation. Fault simulation (FS) is performed after a test pattern has been generated; it identifies all additionally detected faults. After a test pattern has been generated for one fault, the remaining unspecified entries\(^3\) of all input vectors of the test are randomly padded\(^4\), or assigned by using more smart techniques (e.g., additional target TPG [vdL96]). Thereafter, fault simulation is performed to remove the detected faults from the fault list; this process is called fault dropping.

Many (powerful) FSes for sequential circuits have been developed (such as [Nie92b] [Lee92]). However, FS is not within the scope of our research; therefore, we have developed a simple, single fault, single timeframe, single vector, explicit fault simulator. No effort is spend to accelerate this FS, because (our) experiments have shown that the fault simulation computing time is always insignificant compared to the test generation computing time. Section 5.5 describes the sequential circuit FS utilized by the DAT ATPG system.

4) Simulation-based TPG. Deterministic fault-oriented TPG algorithms are known to be time consuming and can be very complex (see Section 1.2). For these reasons, many researchers have suggested simulation based, fault independent test generation approaches [Saa92] [Rud94a]. In a simulation-based TPG (SimTPG) the value of the generated vectors is determined by fault simulation.

In combinational circuits, SimTPG often consists of random generation of input vectors, which are evaluated by a fault simulator. Many faults can be detected that way, because often a fault is detected by many different test vectors; i.e., the probability that a randomly chosen vector will detect a fault is high for many faults.

However, in sequential circuits, random techniques are much less fruitful; the number of test sequences that detect a certain fault is (relatively) low. Therefore, weighted random [Sch75], directed search [Agr88], genetic-algorithm based [Rud95], and other SimTPGs have been proposed which attempt to increase the probability that a "randomly" generated test detects many faults. Figure 3.5 shows the flow diagram of most (sequential circuit) SimTPGs. In these TPGs, test vectors are selected from candidate test vectors. These candidate vectors are initially random vectors (box A in the figure). After FS of the candidate vectors, the "best" vectors (based on cost functions) of the candidate vectors are

---

\(^3\) Usually not all primary inputs have to be assigned to test a fault.

\(^4\) I.e., setting unspecified entries of the input vector to randomly chosen 0s or 1s.
selected and added to the test (box $B$). A cost function indicates the suitability of a (candidate) test vector and directs the search in the test generation process; low costs implies a better suitable test vector. The cost-functions are usually based on heuristics [Agr88], Genetic Algorithms (GAs) [Rud94b] or simulated annealing [Cor97]. The selected test vector is simulated in order to identify the detected faults. New candidate vectors are derived from the original candidate vectors by evaluation of the cost functions, and applying bit-flipping [Agr88], or selection (or reproduction), crossover and mutation [Saa92] [Rud94b], or other methods [Sch75] [Wun90] (box $C$). The SimTPG usually stops when the progress in fault coverage has been stopped or is too low.

The main advantage of SimTPG is that in a relatively short computing time, test patterns are generated for most "easily" detectable faults. The main disadvantage of SimTPG is that the TPG is not complete; i.e., it is not guaranteed that a test is found for a testable fault, and hence untestable faults cannot be identified.

5) Learning processes. Various learning processes are used to lighten the task of the (D)TPG process. They can be performed one or more times anywhere during the ATPG process. However, they often are performed after the pre-processes.

One form of learning is based on implication, to learn global/indirect implications which the normal direct or local implication routines can not identify by local reasoning. In [Sch88b], a static learning procedure has been proposed, which is based on the logical inference rule of contraposition. In [Zha97], a learning technique was proposed based on contraposition and set-algebra, which is much more powerful than the original static learning from [Sch88b]; however, it is memory inefficient.

6) Post-processes after TPG. After the ATPG stage(s), usually test set compaction techniques are applied, to reduce the test set size without loss of fault coverage. A commonly used technique is reverse fault simulation: All detected faults are again marked undetected, test patterns are fault simulated in the reverse order of the order they were generated in. Usually, the first generated test patterns do not detect any additional faults and are removed from the test set. A reason why reverse fault simulation is often very profitable (especially for combinational circuits), is that the last generated test patterns were generated for hard-detectable faults (which were not detected by any of the earlier generated test patterns); hence, these test patterns are required. The first generated test
3.2 DTPG for combinational circuits

As shown in Section 3.1 (Page 36), there are three DTPG algorithms which often form the basis for other TPGs, for combinational as well as for sequential circuits: D-algorithm [Rot66], PODEM [Goe81] and FAN [Fuj83]. A description of these classic TPG algorithms can be found, of course in the three papers introducing these algorithms, but also in books such as [Abr90] and [Fuj85]. The FAN algorithm is the most powerful, and usually forms the basis of most current state-of-the-art DTPG algorithms.

In this section, the FAN algorithm and relevant techniques applied in the DTPG of the DAT ATPG system for combinational circuits [vdL96], are described. The STPG method, proposed in Chapter 4 and described in Chapter 5, is based on this DTPG.

Subsection 3.2.1 presents the FAN algorithm. Subsection 3.2.2 describes test generation for 3-state combinational circuits, as performed in the ATPG system DAT [vdL96].

3.2.1 The FAN algorithm for combinational circuits

The fanout-oriented test generation algorithm FAN, which was developed by Fujiwara and Shimono in 1983 [Fuj83], is the most powerful basic algorithm; nowadays, more than 15 years after its introduction, most state-of-the-art TPGs are still based on this algorithm.

FAN belongs to the class of branch-and-bound TPG algorithms. For a given target fault, such a TPG algorithm searches for a solution (i.e., a test) systematically; this involves a decision making and remaking process. Whenever there are multiple alternatives to test the target fault, the TPG has to (heuristically) choose an alternative and try this decision. But if the decision leads to a conflict or inconsistency in the circuit, the TPG has to remake that decision if there is an alternative (branching), or discard it if none are left (bounding) (backtracking). To guarantee that the complete search space; i.e., the complete space of possible solutions, is explored, the decision-making process has to be systematic. The search space is systematically traversed by recording each decision on a decision stack, which can be visualized with the decision tree of Figure 3.6a. Figure 3.6b shows the corresponding decision stack; for each decision, a “remade-flag” keeps track of whether the alternative has been tried (v) or not (¬)\(^5\). The size of the search space increases exponentially with the number of decisions. Limiting the size of the decision stack is therefore very important: Consider the case that a fault will eventually be proven untestable and the decision stack is large (i.e., many decisions have been recorded); then many (alternative) choices have to be tried before the decision stack becomes exhausted.

\(^5\)assuming binary decisions.
Consider the case for a testable fault and an incorrect decision being made in the top region of the decision tree; then all alternative choices due to decisions made after this incorrect decision have to be tried first, before this incorrect decision can be remade\(^6\).

FAN applies various techniques to limit the decision stack-size. These techniques try to increase the number of mandatory assignments. In addition, FAN only allows decisions on fanout-stems; they are the reason why conflicts can occur in circuits. In the next two paragraphs, the techniques to find mandatory assignments and the decision-making process on fanout-stems are described.

**Mandatory assignments.** In FAN, Mandatory Assignments (MAs) are identified by exhaustive implication and unique sensitization.

**Exhaustive implication.** FAN reintroduced the D-algorithm’s exhaustive implication (forward and backward implication). The algorithm starts by inserting the target fault in the circuit and then performs exhaustive implication, to activate and propagate the fault. Figure 3.7 shows a part of a circuit. Target fault \( F \) is on line \( m \). MAs to activate the fault are signal values \( 0 \) in the fault-free circuit on line \( m \), and value \( 1 \) on lines \( b \) and \( v \) to justify line \( m \). Line \( n \) gets signal value \( 0/X \). Line \( v \) is stored in the list of unjustified lines because there is no unique solution to justify that line.

The fault-site (i.e., line \( m \)) has a discrepant value (0/1). The D-front consists of those gates which have a discrepant value (0/1 or 1/0; also denoted as \( \bar{D} \) and \( D \)) on one or more inputs, and a (partially) unassigned value at their output; hence, the D-front (now) consists of the NAND gate driving line \( n \).

**Unique sensitization.** When the D-front consists of a single gate \( G \), unique sensitization is performed: The *dominators* of gate \( G \) are identified; i.e., gates through which each directed path from gate \( G \) to the POs/PPOs has to go. The inputs of the dominators which cannot be reached by gate \( G \) (i.e., are insensitive) are assigned their non-dominating values and all resulting implications are performed. For example, the dominators of the (single) D-front element \( n \) in Figure 3.7 are \( n \) (itself) and gate \( r \). All insensitive inputs are assigned the non-dominating value: Value 1 on lines \( k \) and \( y \), and value \( 0 \) on line \( p \). Exhaustive implication then results in value 0/1 on line \( n \), and lines \( k \) and \( y \) are stored in the list of unjustified lines.

\(^6\)However, techniques exist to skip alternative choices (without loss of completeness of the search) or to rearrange the order of the decisions, such as back-jumping, indexed backtracking, etc.; see e.g., [Mal85].
3.2. DTPG for combinational circuits

Figure 3.7: FAN algorithm example, a test for signal line m SA-1

Figure 3.8: Objective flow from the different objective lists during the multiple backtrace

Decisions on fanout-stems. After all MAs have been set in the circuit, usually the list of unjustified lines is not empty. Furthermore, often the fault-effect has not been propagated to a PO yet. The list of unjustified lines are the Initial Objectives (IO), which have to be satisfied. An objective is of the form \{l, 0req, 1req\}, whereby l denotes a signal line, 0req(1req) are counters, being the number of 0-requests(1-requests) on this line. For IOs the number of requests is always either one 0- or one 1-request. E.g., the unjustified lines of the circuit in Figure 3.7 are converted into IOs: \{v, 0, 1\}, \{k, 0, 1\}, \{y, 0, 1\}. They are inserted into the IO list. To propagate the fault-effect one level of gates closer to the POs, a D-front element \(DE\) is selected (the one closest to POs). An \(X\)-path check\(^\text{7}\) procedure on \(DE\) is performed to see if there is still a directed path of (partially) unspecified signal values from \(DE\) to at least one PO, which can propagate the discrepant value from \(DE\) to a PO. If not, \(DE\) is discarded from the D-front, and the next element is selected. If a suitable \(DE\) has been selected, an objective is inserted in the IO list in order to propagate the \(D/\bar{D}\) value of element \(DE\) through that element.

Figure 3.8 illustrates the IO list creation from the list of unjustified lines and an objective for fault propagation. Given the IO list, FAN starts the Multiple Backtrace (MB) procedure, which heuristically finds a Final Objective (FO). A FO is of the form \{l, 0req, 1req\}, whereby \(l\) is a fanout-stem, and 0req(1req) denotes the value(s) requested on that stem. Based on the 0req and 1req counters a decision is made on this line; the value with most requests is selected initially as the first choice.

In the MB, the IO list becomes the Current Objective (CO) list (arrow 1 in Figure 3.8)

\(^\text{7}\)Introduced by the PODEM algorithm [Goe81].
and this list is maintained by the MB: COs are converted into COs one level of gates lower in breadth-first fashion (arrow 2 in the figure). For example, the $CO_a$ (CO of line $v$) is \{\$v, 0, 1\}. Using the controllability values of the inputs of gate $v$, MB chooses the “easiest” input to justify line $v$. Assume that line $d$ is chosen. A new CO \{\$d, 1, 0\} is stored in the CO list. Another example, the $CO_a$ is justified with two new COs: \{\$f, 0, 1\} and \{\$w, 0, 1\}.

When MB reaches a fanout-stem, the corresponding CO is stored in the Fanout Objective (FanO) list (arrow 3). If the fanout-stem was already in this list, then the value request(s) of the CO are added to the request(s) of the FanO (of this fanout-stem) in the list. For example, the $CO_d$ becomes CO \{\$a, 1, 0\} and is stored in the FanO list (line $a$ is a fanout-stem); the $CO_e$ \{\$e, 0, 1\} becomes $CO_a$ \{\$a, 0, 1\}. The FanO is already in the list, therefore the $CO_a$ is added to FanO: \{\$a, 1, 1\}. CO$_f$ results in FanO: \$a, 1, 2\).

When the CO list becomes empty, MB starts handling the objectives in the FanO list. Each objective is removed from the FanO list and one of the following three actions is performed (in that order):

1. The FanO has zero- as well as one-requests (i.e., 0req > 0 and 1req > 0): The MB procedure stops and returns this objective as FO (arrow 4). For example, FanO$_a$ is returned as FO by the MB. Value 1 is decided on line $a$. After exhaustive implication of this decision, the MB continues handling the CO/FanO list (arrow 9).

2. The signal line corresponding to the FanO (fanout-stem) is a headline. Headlines are lines in the circuit that can be set conflict-free to a desired value by a set of PIs, irrespective of other signal values required in the circuit. Line $g$ in Figure 3.7 is a headline, because justification of line $g$ (in terms of assignments on PIs) can never cause a conflict. The advantage of headlines would be that there are fewer headlines than PIs, possibly saving unnecessary (re)decisions, and they are located deeper in the circuit, saving the backtrack procedure some steps. However, the usefulness of headlines is very limited: In most cases only PIs are headlines (because these PIs feed fanout-stems).

If the fanout-stem is a headline, then the corresponding FanO becomes a Head-Objective (HO) and is stored in the HO list (arrow 5).

3. The FanO has either zero- or one-request (i.e., 0req > 0 EXOR 1req > 0). The FanO becomes CO(s) on the input(s) of the from-element of the fanout-stem (arrow 6).

When the CO list and the FanO list are empty, MB finishes and returns the first objective in the HO list as FO (arrow 7). The FO is decided as described before. After exhaustive implication of the decision and unique sensitization for fault propagation, the MB starts again for the next FO in the HO list (arrows 7a). The CO-, FanO- and HO-lists are cleared when during the implications the D-front and/or the list of unjustified lines has been changed; a new IO-list is created (arrow 8).

An alternative of the MB is the single backtrace (SB) which has been introduced by PODEM. The SB procedure finds one FO given one IO. It traces back, in depth-first fashion, through elements, thereby selecting one CO at an element input, given the elements output’s CO. In Figure 3.9, at line $a$, the value 0 is requested; the IO = \{\$a, 1, 0\}. The MB
justifies this objective in a single pass, giving four FOs on the inputs of the circuit (see Figure 3.9a). Using SB, only one CO at a time can be created. Hence, to justify the IO on line a, the new CO is an unspecified input of the NAND that is “hardest” to control at 1 (using the controllability values). If the “hardest” input cannot be justified to the desired value, no effort is spent justifying easier to control inputs. Four backtraces from line a to the inputs are required to justify line a (see Figure 3.9b).

Usually the MB is more efficient and more powerful than the SB; conflicting requirements on fanout-stems are detected by the MB and a decision is made on those stems. This is not possible with SB. However, the MB can result in useless objectives, which can cause unnecessary decisions. Figure 3.10 shows two unjustified values on signal lines c and d. Assume that, during MB, the CO on line a (\{a, 1, 0\}) is created (i.e., value 0 is requested on line a to justify line c) and the FanO on line b (\{b, 1, 1\}) is created (i.e., value 0 (for unjustified line c) as well as value 1 (for unjustified line d) is requested on line b). These conflicting requirements on line b results in a decision: Assume that value 1 on line b is decided. After the decision has been implied, the MB continues. It is clear that the CO on line a has become useless (i.e., line c can no longer become 0), and the objective on line a may result in unnecessary decisions. We will show in Chapter 5, that TPG for deep circuits (i.e., with many levels), often performs much better when the SB procedure is applied. Therefore, our STPG can apply SB (instead of MB) as a strategy (see Section 5.3).
Figure 3.11: Z-discrepant values on switch outputs can change to Boolean discrepant values on a bus output

3.2.2 DAT: Test generation for 3-state combinational circuits

The DAT TPG is capable of handling 3-state elements, busses and I/O terminals; it employs the $25V$ signal model [vdL96], and incorporates learning and dynamic dominator techniques for ATPG similar to [Sch88b] [Wai90]. It takes into consideration the internal (bus-conflict) and external restrictions (Restrictors) for the (3-state) circuits. In this subsection, the major differences between the conventional FAN algorithm (see previous subsection) and the (FAN-based) TPGs of DAT are described. Most of the differences are related to the handling of 3-state circuits. They consist of:

A twenty-five-valued (25V) signal model. To simultaneously simulate values in the fault-free and faulty circuit, the 5UV signal model (see Section 2.4) is extended to a 25 valued (25V) signal model, by taking all combinations of the five values in the 5UV signal model for the fault-free and a faulty circuit case combined. The 25V signal model allows for separate simulation of the fault-free and faulty circuit.

Three-valued (rather than two-valued) decisions: To guarantee complete search in TPG, certain signal lines are allowed to become $Z$; for instance, the output of switch elements and bus elements. These lines have three decision options instead of two.

Three-valued (rather than two-valued) objectives in the backtrace. In 3-state circuits some signal lines are allowed to have three values ($0$, $1$, $Z$). Hence, output lines of elements can have a request for a $Z$ value, which has to be justified. These unjustified lines are converted to objectives (see Section 3.2.1). Due to the third value ($Z$), the objectives have three counters instead of two, and are of the form \{l, 0req, 1req, Zreq\}; the third counter (Zreq) denotes the number of $Z$-requests.

The unknown, but specified value $U$, which can occur on the signal lines (see Section 2.4), prevents the backtrace procedure from making decisions on these signal lines. This distinguishes this value from the unknown and unspecified value $X$. Mandatory assignments or decisions on signal lines with value $X$ results in that the signal value of these lines change to (partially) specified values (e.g., value $I$, value $0/X$, or value $U$).
3.2. DTPG for combinational circuits

![Diagram](image)

**Figure 3.12:** Decision stack during TPG and bus-padding

**An extended discrepancy front.** In addition to the discrepant values 1/0 and 0/1 in 3-state circuits, discrepant values can have value Z in the fault-free or faulty circuit. Although, Boolean discrepant values are required on POs to allow for fault detection, also lines with Z-discrepant (e.g., Z/0, 1/Z) values can become part of the D-front. Figure 3.11 shows two switches (part of a multiplexer) with Z-discrepant output values. The corresponding signal lines are part of the D-front as long as the output value of the to-element (the bus, in this case) is not discrepant. The figure shows that Z-discrepant values can recombine into a Boolean discrepant value at the bus output; hence the fault can be detected at a PO.

**Bus-conflict detection.** Buses and all other potential conflicting elements (i.e., elements which can cause a (bus-)conflict; see Page 15) are not allowed to be in a conflicting state in the fault-free circuit. During TPG, bus-conflicts are detected and backtracking is performed until all bus-conflicts are removed.

**Bus-padding.** After a test has been generated for a fault, often not all PIs are assigned a value. Usually these unassigned PIs are randomly assigned a Boolean value to increase the fault coverage of this test pattern (random padding). However, this can cause undesired bus-conflicts. Therefore, bus-padding is performed after TPG: For each potential conflicting element (usually a bus) with an unspecified output value, a (random) value is requested on the output. These unjustified values are then justified, using the backtrace procedure of the TPG. New decisions are made on fanout-stems and stored on top of the stack of the latest fault-under-test. Backtracking is possible on these decisions, but also on the decisions made for the latest fault-under-test (see Figure 3.12). This can be required when the potential conflicting elements cannot be assigned a value without conflicts, given the current TPG decisions. If all potential conflicting elements have a specified value and no bus-conflicts are detected, then it is assured that the current test pattern on the PIs cannot cause bus-conflicts. The remaining unassigned PIs, can now be randomly assigned a value without causing bus-conflicts.

**Dynamic dominators.** The concept of Dynamic Dominators [Sch88b] [Sch88a] [vDL96] generalizes the Unique Sensitization (US) procedure of the FAN-algorithm (see Subsection 3.2.1, Page 40). US identifies mandatory assignments when the number of D-front elements becomes one, based on the dominators of this single D-front element.
The procedure proposed by [Sch88b] [Sch88a] [vdL96], which has been called Dynamic dominators Determination (DD), can also identify mandatory assignments when the D-front consists of multiple elements. Figure 3.13 shows an example circuit. The current D-front consists of two elements (gates 1 and 2; they are driven by lines d and e, which have a discrepant value). DD assigns value 1 on line b for fault propagation through gate 3; gate 3 is a dominator of the current D-front. In addition, DD can identify mandatory assignments when multiple (dynamic) dominators are controlled by the same off-path (fault insensitive) signal lines. For example, gates 1 and 2 are dominators of the D-front: The fault-effect has to propagate through gate 1 or 2. Both gates have the same non-dominating value and a common off-path signal line a. Hence, on line a the non-dominating value 1 can be assigned. Also gates 4, 5 and 6 are dominators of the D-front, but gate 6 is blocked. Gate 4 and 5 has a common controlling off-path signal line c. Although, the non-dominating value of gates 4 and 5 are different, the DD procedure still recognizes the value 1 necessary to be assigned on line c; due to the inversion before gate 4, both gates 4 and 5 require the same value on c to become fault-propagating.

Experimental results, published in [vdL96], have demonstrated the usefulness of the above techniques for combinational (3-state) circuit TPG. Therefore, we expect that these techniques will also be successful in STPG for (3-state) sequential circuits.

### 3.3 DTPG for sequential circuits

Many TPG methods for Sequential circuits (STPGs) have been proposed over the past four decades, such as EBT [Mar78], FASTEST [Kel89], and STG [Che90b]. Most of these STPGs act on a gate-level circuit description, and are based on the iterative array model (see Section 2.3) and the stuck-at fault model. The iterative array model has the advantage that combinational test generation algorithms and techniques can be applied in STPG.

In this section these STPG methods are classified into several STPG categorizations.
3.3. DTPG for sequential circuits

Figure 3.14: Fault F detected under single (bold line) and multiple (dashed line) observation time approach

Four STPG categorizations can be considered, however only two of them are described in detail in this section; advantages and disadvantages of these two categorizations are discussed. The four categorizations are:

1. The test generation method: Simulation-based or deterministic. These methods, and the classification of the STPGs into one of them, have been described in Section 3.1. In the remainder we focus on deterministic TPG methods.

2. The underlying combinational DTPG algorithm: D-algorithm, PODEM, or FAN. In Section 3.1, Page 36 the classification based on this aspect has been given. In the remainder we focus on the FAN algorithm, which is the most powerful one.

3. The time observation approach: Single observation time approach, or multiple observation time approach [Pom92a].

4. The time processing direction: Forward time processing, backward time processing or a combination of forward and backward time processing.

The last two aspects are described in more detail in this section: Subsection 3.3.1 discusses the time observation approaches. Subsection 3.3.2 describes the three time processing direction variants, and places the known STPG methods into these three categories.

3.3.1 Time observation approaches

In the literature, two detection definitions for sequential circuits are reported [Pom92b] [Pom92a] [Cho93]: The single observation time approach and the multiple observation time approach. Figure 3.14 illustrates both approaches; it shows five timeframes of a circuit. The initial (power-up) state $S$ is unknown (all $U$); however in the physical circuit the power-up state is a state with 0's and 1's (but we do not know what the state is). This initial state may differ after each power-up of the circuit.

Under the single observation time approach, a fault $F$ is said to be detectable if there exists an input sequence such that the fault-free response sequence is different from the faulty response sequence at a specific timeframe $t$ for every pair of initial (or power-up)
Table 3.1: Classification of STPGs based on the time processing direction and the underlying combinational TPG (D-alg., PODEM, FAN or other/none)

states \( S \) and \( S^F \) of the fault-free and faulty circuits, respectively. Often the definition of this approach is further restricted such that it requires that the fault-free response be different from the faulty response at timeframe \( t \) on a specific primary output [Pom93b]. This means that, independent of the power-up state of the circuit, it is known at which timeframe \( t \) (and at which PO) the faulty response value can be observed, in case the fault is present. For example, the test sequence generated for fault \( F \) in Figure 3.14 under the single observation time approach results in fault activation in timeframe 3 and fault detection/observation in timeframe 5 at PO A (bold line in figure), independent of the initial state.

Under the multiple observation time approach, which has been mainly developed by Pomeranz et al. [Pom92b] [Pom91b] [Pom92a] [Pom95], a fault \( F \) is said to be detectable if there exists an input sequence such that the fault-free response sequence is different from the faulty response sequence at some timeframe for every pair of initial states \( S \) and \( S^F \) of the fault-free and faulty circuits, respectively; i.e., the values in the response sequence are dependent of the initial state of the circuit; so the timeframe \( t \) where the fault-free response is different from the faulty response is unknown; it depends on the initial state. However, if the fault is present, the fault is detected at one or more POs, in at least one timeframe of an at fore-hand known set; hence responses of a set of timeframes has to be checked to their fault-free responses, to detect the fault. For example, the fault \( F \) in the figure is detected under this approach in (e.g.,) timeframe 2, 4, or 5 (dashed lines), depending on the initial state. The disadvantage of this approach is that for a single fault with corresponding test sequence, multiple fault free responses are possible (depends on the initial state), and multiple observations of the response (at different timeframes) of the circuit are required to guarantee detection of the fault. Most published TPGs assume the single observation time approach, and so will we do in the remainder of this thesis.
3.3.2 Time processing directions in TPG

A classification of STPG methods can be made based on the time processing direction during test generation (See Table 3.1):

- **Forward Time Processing** (FTP): The STPGs based on FTP generate vectors of a test sequence in the order in which they are to be applied for testing. Column 2 in Table 3.1 shows STPG methods based on FTP.

- **Reversed Time Processing** (RTP): The STPGs based on RTP generate vectors of a test sequence in the reverse order in which they are to be applied for testing. Column 3 in Table 3.1 shows STPG methods based on RTP.

- **Forward propagation, reverse justification time processing**: The STPGs based on the Combined time processing approach (CTP) generate vectors using the FTP approach during fault propagation, and using the RTP approach for state justification. Column 4 in Table 3.1 shows STPG methods based on CTP.

The STPG methods shown in columns 2, 3 and 4 in Table 3.1 can also be classified based on the underlying combinational TPG (see column 1). For example, Essential is an STPG method which is based on the FAN algorithm and applies the RTP approach.

In the next three paragraphs, the three time-processing approaches are described and known STPG methods are classified into one of these approaches (illustrated in Table 3.1).

![Diagram of FTP based STPG generation](image)

**Figure 3.15: FTP based STPGs generate vectors in the order in which they are to be applied for testing**

**Forward Time Processing.** Column 2 of Table 3.1 shows the STPGs applying the FTP approach. Simulation-based TPGs (SimTPGs) (see Section 3.1, Page 37) employ the FTP approach by default; they generate and simulate one vector before the next one is generated for the next timeframe (see Figure 3.15). Two SimTPGs are CRIS [Saa92], which uses genetic algorithms to steer the generation process, and Contest [Agr88], which uses cost-functions to determine the “usefulness” of vectors. FASTEST [Kel89] is an STPG based on PODEM. Starting from the initial state a test sequence is generated for a target fault. FASTEST tries to predict (using heuristics) for a target fault, the number of timeframes required to activate the fault and the number of timeframes required to observe the activated fault. This helps to avoid unnecessary search in the input space.

The advantage of FTP is its unidirectional approach; it makes the STPG algorithm less complex. The major drawback of FTP is that the methods are usually incomplete in
search; i.e., they cannot identify untestable faults. The methods require prediction techniques based on heuristics or other approaches (genetic algorithms), and therefore cannot deterministically identify undetectable faults. However, published results [Rud94a] demonstrate that often tests for many "easily" detectable faults are generated in a relatively short computing time.

![Diagram](image)

**Figure 3.16:** STPGs based on RTP, generate vectors in the reverse of the order in which they are to be applied

**Reversed Time Processing.** Column 3 of Table 3.1 shows the STPGs which employ the RTP approach. Many STPGs have been developed based on the RTP approach: D-algorithm based STPGs: EBT [Mar78], STG1 [Mal85], and BACK [Che88]; PODEM-based STPGs: HITEST [Rob83] and STEED [Gho89]; FAN-based STPGs: Essential [Sch89a] and DUST [Gou93].

All RTP based STPGs generate a test sequence starting in the timeframe with a fault-sensitive or discrepant PO, going back in time, until the initial state has been reached (see Figure 3.16). A so-called Topological Path (TP) [Mar78], which is a potential (fault-)sensitive path, is selected from the fault site in a certain timeframe \( k \) to a potential sensitive PO in timeframe \( k + n (n \geq 1) \), which is to be the last timeframe in the sequence. The STPG starts generating a test sequence in reverse order, starting from the (potential) sensitive PO and back to the fault site [Mar78], and then traces back further until all requirements have been justified (see Figure 3.17). The disadvantage of this approach is that a path has to be selected a priori, explicitly from the fault-site to a PO; the number of possible paths can be very large. In STG1 [Mal85] this was partially solved by selecting only a sub-path at a time; all sub-paths concatenated do form the propagating path for the fault. The STPG method BACK [Che88] gives the STPG much more freedom by selecting only a PO a priori with corresponding discrepant value (1/0 or 0/1); no path is selected a priori anymore. The main differences between the other RTP STPG methods (Essential, DUST, HITEST and STEED) can be found in the underlying basic TPG algorithms. The following three STPG methods, HITEST, STEED, and Essential, have some additional characteristics:

- HITEST [Rob83] allows the user to give extra information to constrain/support the STPG; e.g., illegal states and initialization sequences.
- STEED uses the PODEM algorithm to determine the (partial/complete) ON/OFF sets\(^8\) of all flipflops. These sets are used during STPG; mandatory PI values to justify the values on the assigned state lines are determined by intersection of the ON/OFF sets of these state lines.

Essential, an STPG based on the RTP approach, applies forward processing within timeframes, to avoid a priori determination of a path to be sensitized, or of a PO to which

---

\( ^8 \)A set of PI/PPI vectors that set a specific line ON (i.e., to value 1) or OFF (i.e., to value 0).
3.3. DTPG for sequential circuits

Figure 3.17: RTP-based TPG has to determine a priori the sensitive PO, and/or topological path

The fault-effects must be propagated. However, Essential has to determine a priori the FF(s) from which the fault-effect(s) have to be propagated to a PO. An advantage of using forward processing within timeframes is the possibility of applying powerful techniques for fault propagation, such as unique sensitization [Fuj83], often leading to the identification of extra mandatory assignments.

DUST is based on the BACK algorithm; in addition, it uses a set-based signal model and applies several techniques (such as storage and learning of illegal states) to improve the process of sequential circuit test generation.

The main advantage of RTP is the same as for FTP: Its unidirectional approach; it makes the STPG algorithm less complex. On the other hand, sophisticated heuristics are required to choose the “best” PO (to become discrepant) or sensitive path. These choices have to be stored on a decision stack to allow for backtracking to the next “best” PO or sensitive path. The requirement of extra decisions on top of the usual decisions based on controllability and observability, is an extra source for conflicts; heuristic choices cannot be perfect. In addition, proving a fault to be untestable, implies that it must be proven to be untestable for each of the topological paths (EBT) or to each of the POs (BACK). The numbers of especially paths but also of POs can be very large. Another drawback of the RTP approach is that various efficient forward propagation techniques (unique sensitization, dominators), which are very useful in combinational circuit TPG, are not applicable.

Figure 3.18: STPGs based on the Combined approach (CTP) generate vectors using the FTP approach during fault propagation, and using the RTP approach during state justification

Forward propagation, reverse justification time processing (CTP). Column 4 of Table 3.1 shows the STPGs applying the CTP approach. Nowadays most state-of-the-art STPG methods employ this combined approach. STPG using the CTP approach consists of two phases: The first is the FTP phase in which the fault is activated and propagated to
a PO (arrow 1 in Figure 3.18). The second phase is the RTP phase for justification of the state requirements determined in the first phase (arrow 2 in the figure).

An STPG which uses two directions is Stallion [Ma 87][Ma 88]. However, this STPG does not use the test generation algorithm for backward state justification. FTP is used for fault activation and propagation, and (backward) state justification is performed using a (partial) state transition diagram (STD). Stallion assumes that the fault is present only in a single timeframe (because the STDs are only generated for the fault-free circuit). They claim that by ignoring the fault in the other timeframes (during STPG), less than 5% of the generated test sequences do not detect the target fault (as is shown during FS when the fault is assumed to be present in each timeframe). A disadvantage of Stallion is the large memory overhead of storing the STD (the STD grows exponentially with the number of FFs of the circuit); therefore, Stallion cannot handle large circuits. In addition, this TPG can only prove faults to be untestable, if they are combinationally untestable; i.e., untestable in one timeframe.

HITEC [Nie91] was the first true CTP based STPG, using deterministic test generation for both directions. The PODEM algorithm is used for line/state justification and fault propagation. FOGBUSTER [GV95] and MOSAIC [Dar97] are based on the FAN algorithm, which is more powerful.

In the past, two drawbacks were considered to be associated to the CTP approach:

1. The STPG has to be capable of performing test generation in two directions, which makes the STPG method "very complex" [Mal85]. However, many STPG methods (e.g., HITEC, MOSAIC) have proven that performing STPG in two directions needs not be an unsurmountable problem. Actually, it is not a disadvantage, but an advantage, because all techniques developed so far for combinational circuit TPGs are usable for STPG; techniques developed for justification of requirements (e.g., forward and backward implication, multiple backtrace (see Page 41)) can be applied in STPG as well as techniques for fault propagation (e.g. unique sensitization (see Page 40), dynamic dominators (see Page 45)). The techniques usually aim at finding mandatory assignments to test the fault. More mandatory assignments may lower the number of decisions to be made and thus, reduce the probability that incorrect decisions are made. STPG methods based on RTP or FTP only, can apply a limited number of these successful techniques for combinational TPG.

2. The memory requirements for STPG can become very large, because all timeframes during test generation for a fault have to be stored in memory [Mal85] [Che88]. Although all generated timeframes during test generation for a fault have to be stored in memory (to assure completeness in search), for most circuits the memory usage is not too large; HITEC, MOSAIC and FOGBUSTER have reported experimental results for large sequential circuits, which show that their system can handle those circuits. However, there are also sequential circuits which require very long test sequences, and hence, do require large amounts of memory. To handle those circuits successfully, techniques have to be applied to save on memory usage, e.g., by using compressed vectors (see Section 7.4). Note however, that most current computers have enormous amounts of memory, so very large circuits can be handled
and very long test sequences can be generated. Moreover, virtual memory ("swapping") combined with careful STPG design/implementation (by trying to stay inside a timeframe as long as possible) eliminates the suggested problem entirely.

3.4 Summary and conclusions

In this chapter, the ATPG system structure for combinational as well as sequential circuits has been presented. In the first part of this chapter, the main components of the ATPG system have been identified and presented. These components are the pre-processes, the simulation-based TPGs, the deterministic TPGs, the fault simulators, the learning processes and the post-processes.

In the second part of this chapter, one main component has been considered in detail: Deterministic TPG (DTPG). A detailed description of the FAN-algorithm for combinational circuits has been given. This algorithm generates tests for SA-faults, by alternately identifying and implying mandatory assignments (applying exhaustive implications and unique sensitization), and making heuristic decisions on fanout-stems. The relevant methods and techniques applied in the FAN-based DAT-TPG have been introduced: The 25V signal model, three-valued decisions and objectives, bus-conflict detection and bus-padding, and dynamic dominators. Deterministic TPG for sequential circuits has been considered by defining four STPG methodologies: Based on the test generation method (simulation-based or deterministic), based on the underlying combinational DTPG algorithm (D-algorithm, PODEM and FAN), based on the time observation approach (single observation time and multiple observation time), and based on the time processing direction (FTP, RTP and CTP).

Existing STPG methods have been classified into several STPG categorizations, and shortcomings, disadvantages or advantages of these methods have been discussed. It has become clear that current state-of-the-art (deterministic) STPG methods are based on the FAN algorithm and the CTP approach.
Chapter 4

Problems in STPG and proposed solutions

In this chapter unsolved problems in TPG for sequential circuits are identified, together with prior art on these problems, resulting in a list of open questions. To answer these questions, techniques and processes for ATPG for sequential circuits, which have been developed during our research, are proposed.

The main target of our research is to reduce the time-complexity of TPG for sequential circuits, as has been described in Chapter 1. In addition, three problems have been identified in current (state-of-the-art) STPGs: incomplete and inefficient search in STPG, pessimistic bus-conflict detection, and over-specified Illegal States (ISes). Previous work related to these problems is discussed and a study has been performed on state-of-the-art STPG methods. To meet the main target and to solve the three problems, a new STPG algorithm is proposed, which is based on the FAN algorithm, using a new power-set based signal model. This algorithm applies the CTP approach, with techniques from the combinational DAT-TPG as well as from existing STPG methods, and is extended with new techniques to handle sequential circuits efficiently. In addition, techniques are proposed to perform IS identification to increase the known illegal state space, amongst others by removing IS over-specification.

The framework of this chapter is as follows: Section 4.1 presents unsolved problems, which have been identified during our research. These problems are related to the signal model and IS identification. Therefore, in Section 4.2, prior art on signal models is discussed, and in Section 4.3, prior art on illegal state identification is discussed. Section 4.4 describes existing STPG methods together with their advantages and disadvantages. Section 4.5 summarizes the identified problems, which together with the prior art, result in a list of open questions. Three proposals are presented to answer these questions. Section 4.6 summarizes and concludes this chapter.
4.1 The problems

In this section, problems in STPG which are necessary to be solved for successful sequential circuit test generation, are identified. In test generation for synchronous sequential circuits, many significant contributions have been made, as shown in Chapter 3. Still the fault efficiencies achieved by STPG often are low, in spite of the huge consumed computing times; i.e., the *successfulness* of the STPG is still low. Until now, nobody has succeeded in reaching 100% fault efficiency for all circuits in the ISCAS’89 benchmark set [Brg89]. Especially for larger circuits, STPG results are disappointing, illustrating the complexity of STPG, as described in Section 1.2. Also a *correct* STPG is required; i.e., a testable fault should not be marked untestable or vice versa (many published STPG methods can report incorrect results [Che93]).

Two aspects of STPG which have a large influence on the successfulness and correctness of STPG are: The applied *signal model* and the identification of the *illegal states*. The next three subsections present three problems related to these two aspects. Subsection 4.1.1 and Subsection 4.1.2 show how the correctness of STPG is sometimes violated, by incorrect *bus-conflict detection* and *incomplete search*, mostly due to shortcomings of the applied signal model. In addition, Section 4.1.2 also shows the *inefficient search* of enumerative signal model based STPGs. Subsection 4.1.3 describes the concept of illegal state space identification and the problem of *over-specified* ISes, which are encountered during STPG.

4.1.1 Bus-conflict detection and the applied signal model

In Section 2.1, 3-state circuits have been introduced. Current industrial circuit designs usually are 3-state circuits; these circuits contain 3-state elements, such as buses and switches. As has been explained in Section 2.1, Page 15, it is not allowed to simultaneously drive a 0 and a 1 to a bus; such a situation is called a *bus-conflict*, and may damage the circuit. Therefore, test patterns should not cause bus-conflicts in fault-free circuits during production test. Various published TPG methods do guarantee the generation of bus-conflict free tests [Ita86b] [vdL94b] [Woh96] [Dar97]; however, often the TPG is *pessimistic*: It may erroneously signal bus-conflicts. Figure 4.1 shows a correctly assumed bus-conflict (a) and an incorrectly assumed bus-conflict (b). Because the switches in Figure 4.1b have the same data value (0), the output value can only become 0 or Z, and a bus-conflict can never occur. Many logic simulators incorrectly assume a bus-conflict in such a case (e.g., [Dar97] and verilog-XL [Cad97] based simulators)\(^1\). This can result in

---

\(^1\)Note that the SUV signal model with bus-patch (see Section 2.4) does not have this problem.
4.1. The problems

![Decision Diagram](image)

**Figure 4.2:** Two decision options on PPIs lead to incomplete search in STPG; fault f is declared untestable

A lower fault coverage, because faults may be incorrectly marked untestable.

4.1.2 Complete and efficient search and the applied signal model

In (S)TPG, decisions are made during the construction of a test for a fault. A decision usually involves assignment of a value on a signal line. These decisions are stored on a decision stack to allow for backtracking to remake incorrect decisions (see Section 3.1). The decision-remaking on the signal lines (usually (P)PIs) is carried out in order to guarantee completeness of search, and to prevent the backtrace procedure from making the same conflicting decisions on these lines repeatedly. In combinational Boolean circuits, each signal line can assume value 0 or 1. So for each line, two decision options are possible (0 and 1). Complete search in TPG is guaranteed by using these two decision options per signal line for Boolean combinational circuits. However, sequential Boolean as well as 3-state circuits often contain signal lines which can assume, in addition to the values 0 and 1, the values Z and/or U: Signal lines which drive 3-state elements are allowed to become Z, hence these lines have three decision options (0, 1, and Z). The *unknown*, but specified signal value U can be assigned to various sorts of signal lines (see Section 2.4): outputs of 3-state elements (e.g., the output value of a bus is U in case of a bus-conflict), signal lines driven by PI s which have been fixed to U (Restrictor [Kor93]), and signal lines driven by uninitializable FFs. The value U prevents the backtrace procedure from making decisions on signal lines with this value. This distinguishes value U from the unspecified value X.

In addition to the purposes of value U mentioned above, signal lines driven by (non-scannable) FFs may become U after backtracking, when no reset-state is considered, in order to guarantee complete search in STPG [Che93]. Hence, these lines have three decision options (0, 1, and U). Many STPGs methods [Che88] [Sch89a] [Nie91] are incomplete in their search because they ignore this third decision option value U. To illustrate the necessity having a third decision option (U) on PPIs for complete search in STPG, Figure 4.2 shows an decision-making example leading to incomplete search. Assume that the PPIs have two decision options (0 and 1). Initially state \{0, 0\} has been decided to detect the fault. However, assume that state \{0, 0\} can not be justified, so backtracking is

---

2These FFs can not be initialized to a Boolean value, due to oscillation.
performed; the decision on PPIa is remade: The state becomes \{1, 0\}. The PI (see Figure 4.2) is assigned value 0 and the fault is again detected on the PO. Up till now STPG performs correct. However, assume that also state \{1, 0\} is unjustifiable, and backtracking continues. There are no decision options anymore on PPIa, therefore the decision on PPIa is discarded and the next decision (PPIb) is remade, resulting in state \{X, 1\}. With this state the fault-effect is blocked, and backtracking continues, eventually (possibly incorrectly) declaring the fault untestable. The use of two decision options per PPI is based on the assumption that when, for example, state \{0, 0\} and state \{1, 0\} are unjustifiable (illegal), also state \{X, 0\} is unjustifiable. This assumption is incorrect: State \{X, 0\} is not necessarily unjustifiable; in the example of Figure 4.2, the fault is detected when the state is \{X, 0\} and the PI has value 0; however, state \{X, 0\} has not been tried by the STPG. Hence, the use of two decision options per PPI results in incomplete search in STPG.

The underlying reason for this incompleteness is an over-specified state; i.e., one or more lines driven by FFs (i.e., by PPIs) are assigned a value, which does not contribute to the detection of the fault-under-test. It is clear that value 1 on PPIa (after the remake; see Figure 4.2) is not essential to activate and/or propagate the fault (often remade values do not contribute to the test for the fault, because they are actually random assignments), and hence, the state is over-specified. However, the STPG will (thus unnecessarily) try to justify the state \{1, 0\}. This can result in longer test sequences, or even worse, in incorrectly declaring a fault untestable (as shown in Figure 4.2).

PIs do not have the problem of incorrectly declaring a fault untestable by over-specification, because they will eventually always get a value 0, 1 or Z; i.e., they will become specified. For PPIs this is not true, because we consider that the circuit starts in an unknown unspecified state. Thus during the (backward) justification process the PPIs have to become value X eventually (i.e., no requirements on the PPIs).

Figure 4.3a shows that the use of the third decision option \(U\) on the PPIs solves the problem of incomplete search of STPG (as has been shown in Figure 4.2). Due to the decision option \(U\) on PPIa, also state \{\(U, 0\)\} is decided (via backtracking) on the PPIs and thus state \{\(X, 0\)\} is tried to be justified (see Figure 4.3a).

Although complete search is guaranteed in that way, over-specified circuit states can

---

\(^3\) Value \(U\) on a PPI becomes value \(X\) on the corresponding PPO of the previous timeframe.
still occur, due to the decision-making process: After state \( \{0, 0\} \) (see Figure 4.3a) has been proven to be unjustifiable, the over-specified state \( \{1, 0\} \) is tried (value 1 on PPla is not essential). This state is unnecessarily being justified, making the test sequence probably longer than required (if state \( \{1, 0\} \) is justifiable). This (over-specification) problem can be solved by changing the order of decision-making on PPIs [Che93]: From \( B \rightarrow \bar{B} \rightarrow U \) to \( B \rightarrow U \rightarrow \bar{B} \ (B \in \{0, 1\}) \). Figure 4.3b illustrates this last decision-making order: After state \( \{0, 0\} \) has been proven unjustifiable, first state \( \{U, 0\} \) (which cannot be over-specified) is being justified, and then eventually state \( \{1, 0\} \) (which also cannot be over-specified now, because STPG already proved that state \( \{U, 0\} \) prevents detection of the fault).

To conclude, a third decision option solves the problem of incomplete search in TPG for sequential Boolean as well as 3-state circuits. In addition, over-specified states cannot occur when a decision-making order of \( B \rightarrow U \rightarrow \bar{B} \) is applied.

However, this decision-making process, based on two alternative decisions per decision (e.g., initial decision 0 on PPla can be remade to either value 1 or value \( U \)) is inefficient in terms of backtracks. Figure 4.4 shows the decision tree of the circuit of Figure 4.3b. If (eventually) PPla has to become 1 instead of the initial decided value 0, two backtracks are required at least. After remaking value 0 to value \( U \), many other decisions on the (P)PIs may be made; these decisions have to be remade too, before the decision on PPla is allowed to be remade to value 1 (see Figure 4.4). The underlying reason for this inefficiency is the multi-valued\(^4\) enumerative signal model. Such a signal model implies that when an initial decision on a signal line fails, \textit{one} of the other possible decisions options is assigned to this signal line; which decision option is rather arbitrary (however, above we argumented that in the case of three-valued logic (0, 1, and \( U \)) the decision-making order of \( B \rightarrow U \rightarrow \bar{B} \) is the optimal one). It is not possible to "assign \textit{all} other possible decision options simultaneously" to this signal line; i.e., in our example, remaking decision 0 on PPla to the assignments of value 1 and value \( U \) simultaneously, which would properly reflect that it cannot be 0.

\(^4\)Here this means: More than 2 (as is the Boolean case).
Figure 4.5: Circuit with n FFs including a BCD-counter has 2 very sparsely assigned GISes

4.1.3 Illegal state space detection and over-specified ISes

During STPG, entering into illegal parts of the search space (i.e., parts that cannot result in a test pattern (for the fault-under-test)) has to be detected as soon as possible by recognizing:

- Inconsistent requirements to detect the fault. E.g., when a 0 and a 1 are required on the same signal line, backtracking has to be performed.

- An empty D-front: The fault cannot be propagated to a flipflop-input (PPO) or PO, given the currently made decisions.

- A state is required which cannot be justified; i.e., no initialization sequence exists for that state. Such a state is called an illegal state (IS) (an illegal state is called global illegal (GIS) if the state is illegal in the fault-free circuit, because it then holds for the global TPG process, and not just for TPG of one fault).

Identification of (G)ISes is important because often many states cannot be reached with an initialization sequence; i.e., they are (globally) illegal. For example, 6 out of 16 states of a BCD-counter cannot be reached and are globally illegal. Often the number of possible states (which is doubled each time a flipflop is added to the circuit) is much higher than the number of functionally used states; i.e., many states are globally illegal. The ratio between the functionally used states and the state universe is called the density of state-encoding. In [Mar95], a low density of encoding is reported as the major cause for the high complexity of STPG (see also Section 1.2). In our experience with the ISCAS'89 [Brg89] benchmark circuits as well as industrial 3-state circuits, the circuits generally have a low density of encoding. This is indicated by the fact that the GISes are often sparsely assigned, especially when there are many FFs; i.e., the GISes have only a few assigned entries compared to the total number of entries. Consider the BCD counter, which has 6 fully assigned GISes out of 16 states. These 6 GISes can be covered by 2 GISes: 1X1X and 1LXX. Figure 4.5 shows a circuit with a large number of FFs (n); 4 of them are part of a BCD-counter. Due to the BCD-counter, the circuit has two GISes with only 2 entries assigned, which each cover at least 25% of the state space; i.e., at least 25% of the state space is global illegal; in the example of course 6/16th part of the total state space is illegal due to the BCD counter.
Figure 4.6 gives an impression of how the STPG search space (within one timeframe) is formed by the state space combined with the PI space. The state space can be divided into a fault independent and a fault dependent space as shown, corresponding to the fault-free and the faulty circuit. GISes that have been found, form a space named the known GIS space. Figure 4.6 shows a (non-contiguous) GIS space by shaded areas in the state space, with the known parts dark-shaded. It is very important to know an as large as possible part of the GIS space during STPG because it helps to constrain the state space to justifiable states only. As soon as a state inside the known GIS space is reached, STPG can perform backtracking. This avoids fruitless justification attempts, saving computing time. Because GISes are fault independent, they can be used throughout the STPG process for all faults. Therefore, increasing the known GIS space, by identification and storage of GISes, will often result in increased fault efficiency after STPG, and/or decreased computing time.

(GI)Ses are (naturally) found during STPG when justification of some (fault-free) portion of the state fails. This means that the decision stack was exhausted during backtracking without reaching a solution, and does not include aborted justification. Thus, upon justification failure, the justification process has proven that a certain state is unjustifiable, hence it is an IS. Many ISes can be found during the actual generation of tests. However, the ISes found in this way are rarely as sparsely assigned as we argued above that they can be (i.e., they are over-specified, so that (too) many, less effective, (G)ISes are stored, preventing maximal profit, and possibly causing storage problems.

4.2 Prior art in STPG, related to signal models

Nearly all STPG methods use a signal model based on two values \( \{0, 1\} \) or on four values \( \{0, 1, Z, U\} \) to denote the value of a signal line. TPG methods for combinational Boolean circuits require a signal model based on values \( \{0, 1\} \). A signal model based on four values is required for (S)TPGs which handle 3-state and/or sequential circuits (see Section 2.4). The 3-state circuits require the value \( Z \) to express a floating (high impedance) "value", and 3-state as well as sequential circuits require the value \( U \) to express a specified but unknown value. In most signal models, the value \( X \) denotes the unspecified value. Thus Boolean circuits require a 3-valued (3V) signal model \( \{0, 1, X\} \), and 3-state circuits require a 5-valued (5UV) signal model \( \{0, 1, Z, U, X\} \).
Figure 4.7: 9V signal model (b) has more resolution than the 5V signal model (a): the 9V local implications of the SA-0 fault lead to a non-discrepant value on the only PO

To describe the fault-free and faulty circuit simultaneously, the signal models are usually extended into two dimensions. For example, a signal model based on values \( \{0, 1\} \) is extended to \( \{a/b\} = \{0/0, 0/1, 1/0, 1/1\} \), where the composite logic value \( a/b \) indicates that the value in the fault-free circuit is \( a \) and the value in the faulty circuit is \( b \). Based on these signal models, many variants have been proposed. These models are used for TPG for combinational as well as sequential circuits. They often differ from each other in:

- The number of values they employ.
- The memory required to store a value.
- The evaluation speed of the circuit elements.
- Their resolution [Raj90]. A better resolution of a signal model means that the values are more precise; it often reduces the number of backtracks during TPG and often improves the fault efficiency reached for a circuit.

Roth (1966) [Rot66] with his D-algorithm introduced the 5-valued (5V) signal model (\( \{0, 1, D, \bar{D}, X\} \)). Value \( D \) (\( \bar{D} \)) denotes value \( 1/0 \) (\( 0/1 \)) in the fault-free/faulty circuit. Many other TPG algorithms use this model, such as, the PODEM-algorithm [Goe81] and FAN-algorithm [Fuj83]. However, this model lacks resolution: The values on the signal lines are only specified (i.e., they have value \( 0, 1, D \) or \( \bar{D} \)) when the fault-free and the faulty values are known. In all other cases the signal line value is \( X \).

Muth (1976) [Mut76] introduced the 9-valued (9V) signal model to improve this lack of resolution of the 5V signal model. This signal model fully separates the fault-free and faulty circuit values; all combinations with values \( \{0, 1, X\} \) for fault-free and faulty circuit are taken into account. This makes the 9V signal model much more powerful than the 5V model, as is illustrated in Figure 4.7. In the figure it can be observed that 9V local implications (Figure 4.7b) of the SA-0 fault already lead to a non-discrepant value on the only PO, hence the fault is untestable. With the 5V signal model (Figure 4.7a) this is only detected after making and implying at least two decisions. A variant on the 9V signal model is the SPLIT signal model [Che88]. The 9V signal model is used in many TPGs, for instance, in combinational TPGs: SOCRATES [Sch87] and TOPS [Kir87], and in sequential TPGs: HITEC [Nie91], STEED [Gho91] and Essential [Sch89a]. However, the 9V signal model has two major shortcomings:
4.2. Prior art in STPG, related to signal models

1. It is not suitable for TPG for 3-state circuits. These circuits require a signal model based on four values, \( \{0, 1, Z, U\} \).

2. It is not suitable for TPG for sequential circuits, when no reset-state is considered. 9V signal models are based on two values (0 and 1), hence only two decision options per signal line are possible. To avoid over-specified circuit states, which can result in incomplete search in STPG, three decision options are required (see Section 4.1.2). HITEC [Nie91], Essential [Sch89a] and all other 9V signal model based STPGs have this over-specification problem.

Several signal models have been reported, which overcome the first shortcoming, and hence are suitable for 3-state circuits:

**Levin** (1981) [Lev81] introduced a 6-valued signal model: The basic values 0, 1, Z and \( X \), and values 0\(\lor Z \) and 1\(\lor Z \). The specified value 0\(\lor Z \)(1\(\lor Z \)) denote 0 or Z(1 or Z) on a signal line; i.e., the value of this signal line can not be 1(0). This signal model solved the pessimistic bus-conflict detection as shown in Figure 4.1b: The bus inputs and output evaluate to 0\(\lor Z \)(i.e., the signal values are either 0 or Z, but cannot be 1) in this signal model, and no bus-conflict is reported. However, this signal model is only used for logic simulation.

**Ogihara et al.** (1983) [Ogi83] proposed a 14-valued (14V) signal model. It extends the 5V signal model with 5 new signal values \( \{Z, H0, \overline{H0}, H1, \overline{H1}\} \) in order to handle 3-state elements. \( H0 \ (H1) \) describes a signal that is 0(1) in the good circuit and Z in the faulty circuit. \( \overline{H0} \ (\overline{H1}) \) describes a ‘good Z’ and a ‘faulty 0(1)’. The remaining four values are exclusively meant to solve the problem of fault propagation to clock-lines of memory elements. This 14V signal model lacks resolution, like the 5V signal model. For instance, composite values 1/\(X \) and \( X/0 \) are not included.

**Itazaki et al.** (1986) [Ita86a] [Ita86b] improved the resolution of the 14V signal model: Three values \( \{H2, \overline{H2}, Y\} \) have been added, resulting in a 23-valued signal model (remaining 6 new values are exclusively meant to solve the problem of fault propagation to clock-lines). \( H2 \) describes a signal that is \( X \) in the good circuit and Z in the faulty circuit. \( \overline{H2} \) describes a “good Z” and a “faulty X”. \( Y \) models an indeterminate value (0, 1 or Z). Although, the resolution has been improved compared to the 14V signal model, still the resolution is low; signal values in Boolean logic gates are 5-valued.

**van der Linden et al.** (1994) [vdL94b] introduced a 25-valued (25V) signal model for TPG with 3-state circuits. The signal model is comparable to the 9V model but is based on the 5UV signal model. By taking all combinations of 5 values (0, 1, Z, U and \( X \)) for the fault-free and faulty circuit, the signal model is expanded to 25 values. However, also the 25V signal model has a pessimistic view concerning bus-conflicts; this signal model incorrectly assumes a bus-conflict in the circuit of Figure 4.1b. Therefore, the so-called bus-patch [vdL94b] was introduced, as described in Section 2.4. A disadvantage of the bus-patch is that it can be CPU expensive because the values \( X \) and \( U \) on 3-state lines have to be analyzed for their ‘real’ meaning. Also, the bus-patch is not an elegant technique to solve this lack of precision of the 25V system.

To overcome the second shortcoming of 9V signal models, incomplete search during TPG and over-specified states, several signal models have been reported:
Niermann et al. (1991) [Nie91] proposed the STPG method HITEC, which uses the 9V signal model. However, it partially overcomes the over-specification problem on the PPI state by eliminating unnecessary PPI assignments: Each assigned PPI is changed to an $X$, for both the fault-free and faulty circuit, one at the time, and it is verified whether all objectives are still met. If not, then the PPI assignment is restored, otherwise another assignment has been saved. Disadvantages of this scheme are that it does not necessarily result in the minimum set of PPI assignments, it is CPU expensive, and the search process is (still) incomplete (because of the two decision options per PPI, while three decision options per PPI are required (see Section 4.1.2)).

Cheng et al. (1993) [Che93] proposed a 4-valued (4V) signal model ($\{0, 1, U, X\}$) to assure complete search in Boolean STPG. Each PPI can assume three values ($0$, $1$ and $U$). Incorrectly marking faults untestable is not possible with this signal model. In addition, Cheng et al. proposed a decision-making order of $B \rightarrow U \rightarrow \overline{B}$ ($B \in \{0, 1\}$), to avoid over-specified states, and thus avoiding too long generated test sequences. However, this decision-making process can be inefficient in terms of backtracks (see Section 4.1.2). Also, this signal model cannot handle $Z$, hence it is not suitable for 3-state circuit TPG.

Gouders et al. (1993) [Gou93] proposed the signal model CONSEQUENT with three basic values: $0$, $1$ and $U$. The three values are bitwise encoded, such that each signal line has a set of still possible basic values, resulting in a 6 valued signal model. For example, a signal line can have value $\{0, 1, U\}$ (i.e., all 3 bits are set), meaning that this line still can become either $0$, $1$ or $U$.

An STPG based on CONSEQUENT justifies only essential values on the PPIs, and the search of STPG is complete. In addition, this set-based signal model solves the problem of the inefficient decision making of multi-valued enumerative signal models, because remaking an initial decision to all alternative decision options is possible; i.e., the initial decision value is excluded from the set of possible values. E.g., assume that the TPG process decides to assign value $0$ on a signal line. If a conflict occurs and the decision has to be remade, the value of the line becomes $\{1, U\}$; i.e., value $1$ and value $U$ are still allowed on this line. STPG based on the enumerative 4V signal model [Che93] with a ternary decision tree, requires at least two backtracks to change a (conflicting) value $0$ on a signal line to value $1$, if that value is required for the test (see Figure 4.8a). STPG based on the set-based signal model CONSEQUENT requires one backtrack in that case, and makes a (conflicting) value $0$ on a signal line to $\{1, U\}$; after that, value $1$ is deterministically decided by the backtrack procedure and assigned on this signal line, when this is required for the fault-under-test (see Figure 4.8b).

However, CONSEQUENT does not consider the $Z$ value, so it cannot be used for 3-state circuits.

van der Linden et al. (1994) [vdL94b] proposed the 25V signal model (see also 63) and a ternary decision tree for PIs which can drive buses (these PIs are allowed to be $Z$). Although not mentioned in [vdL94b], the 25V signal model can also handle the three values ($0$, $1$, and $U$) as required on the PPIs. Hence, the 25V signal model (with bus-patch) guarantees complete search in TPG (and correct bus-conflict detection, as described on

---

5 In [Gou93] the set $\{0, 1\}$ is not considered, which would denote a line which can only become $0$ or $1$. 
4.3 Prior art in STPG, related to illegal state detection

Figure 4.8: Backtracking, based on enumerative signal models (a) is less efficient than backtracking, based on set-based signal models (b)

page 25). However, also the 25V signal model has disadvantages: The bus-patch can be expensive in computing time, and similar to other enumerative signal models, the 25V model can be expensive in terms of backtracks.

Dargelas et al. (1997) [Dar97] proposed the signal model MOSAIC, derived from CONSEQUENT with four basic values: 0, 1, Z, and U, which are bitwise encoded, resulting in 16 different values. Like in the CONSEQUENT model, the assigned bits of a signal line value denote which basic values are still possible; a value with multiple bits set, is a (partially) unspecified value; i.e., it would be denoted by X in an enumerative signal model. This set-based signal model has the same advantages as the signal model CONSEQUENT. In addition, it considers the Z value, so that STPG for 3-state circuits can be performed. However, the bus-conflicts are not handled properly: MOSAIC is too pessimistic; it incorrectly assumes a bus-conflict in the example-circuit, as shown in Figure 4.1b.

To conclude, no signal model has been reported which simultaneously assures complete search in TPG for sequential 3-state circuits, prevents over-specified states, is efficient in terms of backtracks, and performs correct bus-conflict detection.

4.3 Prior art in STPG, related to illegal state detection

Prior art on illegal states in STPG includes:

Niermann et al. (1991) [Nie91] in their STPG HITEC proposed, GIS identification, naturally performed during STPG, and storage of the GISes to prevent later fruitless justification attempts. Dargelas et al. (1997) [Dar97] performed GIS identification exactly as in HITEC.

Gouders et al. (1993) [Gou93] also performed GIS identification and storage, but in addition, proposed GIS minimization of the identified GISes during STPG. This process

6 Only "correct" in relation to enumerative and set-based signal models. Compared to signal models based on symbolic techniques, even the best enumerative or set-based signal models can sometimes incorrectly assume bus-conflicts.
simply unassigns assigned entries in a GIS one by one, and tries to justify the resulting potential GIS. If justification fails, a new GIS is identified, which covers the GIS being minimized. This latter GIS can be removed.

Methods to obtain more effective GISs before actual test generation starts have been proposed in [Lon95] and [Lia96].

Long et al. (1995) [Lon95] proposed FILL (Find ILLegal states), which is a process to find GISs, using implicit state enumeration based on BDDs. During the process it builds a list of GISs. Because BDDs are used, which for large circuits may be memory consuming while computing them may be CPU expensive, the technique is only successful for small circuits or partitions of large circuits.

Liang et al. (1996) [Lia96] proposed three techniques, using logic simulation of the legal states forward in time, to find the GISs. Starting in the reset state(s) (in the hardest and here assumed case, the state wherein all FF states are unknown), all PI combinations are tried, and all reached (partially) assigned states are stored. From each already reached state, again by trying all PI combinations, other reachable states are found, and so on. If the process terminates, the remaining states are the unreachable, and hence also the unjustifiable states, thus GISs. The big drawback is that when due to time constraints not all PI combinations from all reachable states can be simulated: If not all the reachable states are found, one cannot conclude that the remaining states are GISs. For larger circuits this method will not be useful, since going through all reachable states may be infeasible.

To conclude, very few prior art on IS detection, storage and optimization (e.g., removing over-specified assignments in ISes) before STPG or during STPG has been reported. Practical techniques to explicitly explore the GIS space, by the use of learning processes, have not been reported at all. Also all existing methods identify only fault independent ISes (GISs). However, from Section 4.1 it is clear that identification of the fault independent and dependent IS space is very important and may be a required condition for a successful ATPG system.

### 4.4 Evaluation of STPG methods

In this section relevant previous work on state-of-the-art STPG methods is described; i.e., on the STPG methods based on the FAN algorithm and/or CTP approach, according to the conclusions in Chapter 3. Various useful techniques are described; some of them have been incorporated into our STPG method.

This section is organized as follows: In the next four paragraphs, four state-of-the-art STPG methods are evaluated: HITEC [Nie91], DUST [Gou93], FOGBUSTER [GV95] and MOSAIC [Dar97]. Especially for the STPG method HITEC an extended overview is given, because many aspects of this STPG are also implemented in our STPG. However, we show that all of these STPGs have shortcomings.

**HITEC (T. Niermann; 1991)** [Nie91] has been the first STPG applying CTP. It is based on the PODEM algorithm [Goe81] and uses dominators and mandatory assignments similar to the ones used in FAN [Fuj83], TOPS [Kir87] and Socrates [Sch88b]. A 9-valued
signal model is used, with two-valued decisions; as described in Section 4.1.2 this may violate the completeness in search of the algorithm.

In [Nie91] six techniques have been introduced:

1. Targeted D-front propagation. This technique aims to increase the number of mandatory assignments for fault propagation; i.e., propagation of one or more D-front elements to one or more (P)POs.

   The use of the dynamic dominators of the current D-front has been proven to be very useful for the identification of mandatory assignments for fault propagation during CTPG [vdL96] (see Section 3.2.2). However, in STPG the number of dominators of the D-front often is very small or is zero, because of the large number of D-front elements; often a number of these D-front elements has independent fault propagation paths. Figure 4.9a shows one timeframe of a sequential circuit. Fault-effects are present of fault $F$ in the current, and of fault $F$ in the previous timeframes, resulting in a large D-front (numbered by 1 ... 5 in the figure), which have probably independent fault propagation paths (it is highly likely that paths 1 and 4 in Figure 4.9a do not influence each other), and thus reducing the chance of having dominators (the set of (dynamic) dominators of the D-front is determined by taking the intersection of the dominator sets of each of the D-front elements individually).

   Therefore, in [Nie91] the targeted D-front technique has been proposed: Choose the “best” D-front element (the remaining D-front elements are ignored and stored on the decision stack as alternative decisions) and determine the dominators with respect to that element only, thus increasing the number of mandatory assignments. The “best” D-front element is selected using SCOAP observability measures [Gol80]. Figure 4.9b shows an example circuit. D-front element (driving line) $c$ is selected as “best” D-front, resulting in a mandatory assignment on line $b$ (value 1), which would not be mandatory when the complete D-front was considered during fault propagation (elements $c$ and $e$).

   When fault propagation fails, starting from a chosen D-front element, an alternative D-front element is chosen by removing the top D-front element from the decision stack.

2. STPG is performed within a variable sized window of timeframes. A window
Figure 4.10: Within a window TPG is performed, ignoring the timeframe boundaries.

consists of one or more timeframes. Figure 4.10 shows the iterative array model of a circuit. A window of size four is defined (bold section in the array). TPG is performed within the window, whereby the timeframe boundaries in the window are ignored; i.e., a backtrace is performed over four timeframes, making decisions on PIs of those four timeframes and on PPIs of timeframe \( t \). A (small) disadvantage of this method is the extra memory overhead, because of each timeframe within the window a list of D-front elements and a list of unjustified lines has to be stored. The advantage of this method is that unjustifiable states in the window can be detected earlier, because they are directly tried to be justified by decisions on PIs(PPIs) of timeframes before these states. However, it is to be expected that the number of decisions in one window increases compared to the number of decisions in one timeframe, because each assigned PPI probably results in multiple decisions (on PIs of timeframes before that PPI). Therefore, we expect that this method performs well for highly testable circuits, but not for circuits with many untestable faults.

3. States which are proved to be unjustifiable during STPG are stored as GISes, to avoid wasting computing time in repetitive justifying these GISes. However, we have shown in Section 4.1.3 that this is not enough to have a successful STPG: Optimization of the detected GISes and storage of fault-dependent ISes are necessary.

4. Eliminate unnecessary state specification during the RTP phase. Each assigned PPI is changed to an \( X \) when it is not required for the fault-under-test (see Section 4.2, Page 64). This method partially overcomes the over-specification problem on the PPI state, due to the 9V signal model. However, completeness in search during STPG is still not assured, because the decisions are still two-valued.

5. Using previously generated state sequences to justify states required for the current fault-under-test. Given the current state \( T \), during STPG, which has to be justified. If \( T \) covers one of the previously generated states (e.g. state \( S \); the value of the entries of \( T \) are \( X \) or are equal to the corresponding value of the entry of \( S \)), then the test sequence required for this state \( S \) can be added before the current generated test sequence, to justify state \( T \). Note that the fault may change the reached state \( S \), so that it is not covered anymore by \( T \); therefore, logic simulation of the added test sequence is required.

6. Faults for which test generation has been aborted may be selected as target again when the fault-effects of those faults are propagated to PPOs in one or more previously generated states. This is determined by the FS. Figure 4.11 shows a fraction of the test sequence: \( v10 \ldots v15 \). Fault \( F \) is activated in timeframe \( v12 \) and the fault-effect is propagated to the PPOs of timeframe \( v13 \). This fault is selected as target for STPG and two new vectors are generated and added behind vector \( v13 \), to propagate the fault-effect to a PO. The fraction of the test sequence is determined, which is required to activate the
Figure 4.11: Current test sequence has faults with are propagated to PPOs; Two timeframes are generated by the STPG and added to the current sequence to propagate the fault $F$ to a PO

fault and propagate the fault-effect to the PPO. In our example, the sequence $v_{11} \ldots v_{13}$ will result in an activated fault $F$ in timeframe 12 and a sensitive PPO on $v_{13}$. This part is copied to the end of the test sequence, with the two newly generated vectors added behind this part.

Five of the above six techniques\textsuperscript{7} are also implemented within our STPG. The practical usefulness of these techniques is discussed in the next chapter.

DUST (N. Gouders and R. Kaibel; 1993) [Gou93] is an STPG applying the RTP approach. It is based on the BACK algorithm [Che88]. The main contributions of DUST are, firstly, that it uses a set-based signal model CONSEQUENTIAL (see Section 4.2, Page 64), which results in complete search in Boolean sequential circuits, and in efficient decision-making during STPG. Secondly, GIS identification, storage (as in HITEC), and GIS minimization is performed. The process GIS minimization simply unassigns assigned entries in a GIS one by one, and tries to justify the resulting potential GIS; if this fails, the potential GIS is also a GIS, and covers the GIS being minimized (see Section 4.3, Page 65). The disadvantage of DUST is that it is a RTP based algorithm; usage of powerful fault propagation techniques (such as dynamic dominators determination) cannot be performed.

FOGBUSTER (U. Gläser and H.T. Vierhaus; 1995) [GV95] is an STPG applying the CTP approach and is based on the FAN algorithm. In addition, FOGBUSTER uses several techniques from HITEC. An enumerative signal model based on the values $\{0, 1, U, X\}$ is applied to guarantee complete search in STPG. However, as has been discussed in Section 4.1.2, multi-valued enumerative signal models are not efficient in decision-making, in terms of backtracks.

In [GV95] it is incorrectly claimed that targeted D-front selection is not effective; it would increase the size of the decision stack, which is true; however, especially for testable faults, this technique is very useful, as we will show in the next chapters. Although FOGBUSTER detects state repetitions, the algorithm does not store GISes; i.e.,

\textsuperscript{7}In addition, the missing technique (windows; second technique) is currently being implemented
it only detects fault-dependent state repetitions. This is a large shortcoming of this algorithm.

MOSAIC (A. Dargelas, C. Gauthron and Y. Bertrand; 1997) [Dar97] is an STPG applying the CTP approach and is based on the FAN algorithm. In MOSAIC a 256-valued set-based signal model has been introduced, which is capable to handle 3-state circuits. However, MOSAIC has a pessimistic bus-conflict detection (see Section 4.2, Page 65). MOSAIC performs a FAN based test generation process, using various strategies. Strategies have been introduced for the backtracking method, decision order, fault propagation, and state justification order. The number of aborted faults provides information about the historical efficiency of a strategy. Based on this information, the best strategy is chosen for STPG for the remaining faults.

4.5 Open questions and proposed answers

In the previous sections we have identified three main problems in STPG methods:

1. Many STPG methods are pessimistic with respect to bus-conflict detection in 3-state circuits.

2. Many STPG methods are incomplete in their search; they assume two decision options (value 0 and 1) for PPIs, which can result in incorrectly marking faults untestable. The addition of a third decision option (value $U$) solves this problem; however, then the decision-making process is inefficient in terms of number of backtracks. This increases the computing time for ATPG.

3. Many STPG methods have a lower performance than they could have, because they do not identify and store the (global) ISes, or have identified (and stored) ISes which are over-specified. In addition, often these STPG methods do not apply techniques to expand the known illegal state space.

Hence, to solve the above three problems and to satisfy our target defined in Section 1.4, a state-of-the-art ATPG system for sequential circuits must have at least the following characteristics:

- The system can handle sequential circuits successfully; i.e., reach high fault efficiencies within reasonable time. Therefore, an STPG method is required based on the CTP approach. Only under that approach all sophisticated techniques developed for combinational circuit TPG can be used. In addition, the STPG is based on the FAN algorithm, and stores ISes which have been identified during STPG, to avoid repeated justification of ISes.

The system performs pre-processes and learning processes to support the STPG. These processes include fault collapsing, controllability and observability calculation, (pre-)identification of redundant and untestable faults, and static learning to find indirect/global implications.
4.5. Open questions and proposed answers

- The system can handle 3-state circuits, without pessimism or optimism with respect to bus-conflict detection; i.e., the signal model, applied in the TPG methods, must have sufficient resolution to perform correct bus-conflict detection.

- The system performs complete search during STPG. To guarantee complete search during STPG, three valued decisions are required on signal lines which can become Z, and PPIs which can become U; i.e., the signal model applied in the TPG methods, must include values \{0, 1, Z, U\}. In addition, to guarantee efficient decision-making during STPG, the signal model has to be (power-)set based.

- The system performs learning to find GISes to simplify the STPG task, and performs techniques to identify ISes and expand to known illegal state space (e.g., remove over-specifications in ISes).

These characteristics leads to the following questions which are addressed in this thesis:

1. How to extend the FAN-based ATPG system DAT for sequential Boolean circuits?
2. How to extend the FAN-based ATPG system DAT for sequential 3-state circuits?
3. How to identify ISes during STPG?
4. How to perform identification of untestable faults before STPG?
5. How to perform (static) learning in sequential circuits?
6. How to avoid pessimistic bus-conflict detection?
7. How to assure complete search in STPG?
8. How to avoid over-specified states during STPG?
9. How to perform efficient decision-making during STPG?
10. How to (efficiently) identify the GIS space before STPG?
11. How to expand the known (G)IS space; i.e., “optimize” the stored (G)ISes?
12. How to store ISes efficiently?

To answer these questions the following is proposed:

Questions 1, 2, 3, 4 and 5 will be addressed in Chapter 5, where we describe the DAT ATPG system extended for sequential 3-state circuits. The following ATPG system is proposed:

- A pre-process to identify untestable faults without STPG: Signal lines are identified which cannot be initialized to a certain value, or from which sensitive values cannot be propagated to a PO. The SA-0 and/or SA-1 faults on these lines are untestable.
Chapter 4. Problems in STPG and proposed solutions

- An STPG, based on the combinational circuit TPG DAT. This implies that the STPG can handle 3-state circuits and is FAN based. The iterative array circuit model is used and the STPG is based on the CTP approach, so that all (sophisticated) techniques of combinational circuit TPG can be (re)used, such as multiple/single backtrace, dynamic dominators, and recursive learning [Kun92] [vdL96].

- The State Restrictor (SR) is introduced, which constrains the state space during STPG to legal states, avoiding repetitive justification attempts of ISes.

- Fault dependent controllability and observability cost measures are introduced, because often a large part of the circuit is influenced by the fault.

- Eight STPG strategies to allow for strategy switching during ATPG, depending on circuit characteristics. These strategies are fault propagation before justification of objectives, search order during the backtrace procedure, single backtrace, targeted D-front propagation, find current (PPI) state in previous generated test sequence, ignoring sensitive PPOs, when POs can still be reached in current timeframe, state justification before fault propagation, and recursive learning.

- Static learning, to support the STPG, to learn global implications and fixed lines. We propose to combine the static learning [Sch88b] with recursive learning [Kun92], to find much more indirect (global) implications.

- The STPG has to be capable to use fault simulation information, to select (still undetected) faults which fault-effect is propagated to (a) PPO(s) on certain time-frames.

- Additional target TPG [Pom91a] [vdL96], in order to extend a test for one fault deterministically so that it covers additional faults.

Finally, the memory overhead imposed by the STPG has to be as low as possible, to be capable to handle large sequential circuits.

Questions 6, 7, 8 and 9 will be addressed in Chapter 6, where we propose a new signal model: Power Set Logic (PSL). This signal model overcomes the disadvantages of all signal models discussed in Section 4.2. Using this model, three decision options are available on (P)PIs, to guarantee complete search during STPG. In addition, PSL provides efficient decision-making during STPG in terms of number of backtracks, and avoids over-specified circuit states, because of its ability to express the exclusion of a certain value. Due to this ability, apart from decisions, also all re-decisions on (P)PIs during backtracking are made deterministically, in contrast to the random re-decisions made during STPG based on enumerative signal models. Furthermore, incorrect bus-conflict detection is avoided.
Questions 10, 11, and 12 will be addressed in Chapter 7, where we propose new techniques and processes to perform (G)IS identification, storage and optimization. The techniques use the known GISes to generate candidate GISes, which are then tried to be justified. If this fails; i.e., the justification process proved a candidate GIS to be unjustifiable, the candidate GISes becomes available as a new and better GIS, often replacing previous, less powerful GISes. This way of expanding the known GIS space has the advantage that the number of GISes in the list is reduced, saving memory and computing time, while it covers a larger part of the GIS space. Three new processes are proposed, using one or more of the new techniques:

1. A process before or between ATPG stages, called GIS learning, which is a learning process. It aims at identifying an as complete as possible GIS space. This process can be expensive in computing time. Therefore, the legal state cache is proposed. This cache avoids useless repetitive justification of known legal states during the learning process.

2. A process during ATPG, called Check_Illegal_States (CIS), which removes redundant GISes (i.e., GISes which are covered by other GISes) and finds common fractions between two or more GISes. These common fractions form candidate GISes which are handled as described above.

3. A process between ATPG stages, called Expand_Illegal_States (EIS), which removes over-specified assignments in GISes.

To demonstrate the effectiveness and efficiency of this new ATPG system, experiments have been performed on the ISCAS'89 [Brg89] circuits and a number of industrial circuits. Furthermore, a case study has been performed on the Philips 80C51 micro-controller [Phi97]. This study will be addressed in Chapter 8.

4.6 Summary and conclusions

In this chapter, three problems have been identified which can occur in TPG for sequential (3-state) circuits, and can invalidate the correctness and the successfulness of the STPG: First, due to shortcomings of the applied signal model, the STPG can be pessimistic in bus-conflict detection. Second, due to shortcomings of the applied signal model and the decision-making process, the search of the STPG can be incomplete, or can result in over-specified states, or can be inefficient in terms of number of backtracks. This may result in too high CPU costs, needlessly longer test-sequences or even in faults being incorrectly marked untestable. Third, detection and storage of the illegal state space is essential for a successful STPG. However, often a small part of the illegal state space has been identified during STPG, and the identified ISes are usually over-specified. This over-specification may result in (too) large IS lists, and (needlessly) reduced knowledge of the IS space.

Prior art related to these problems has been studied. We have found no signal model that assures correct search in STPG, and prevents over-specified states during test generation, and performs correct bus-conflict detection. Also, very little work has been published
on IS detection, storage and optimization, before or during STPG. No practical techniques have been reported which explicitly identify the GIS space.

Four state-of-the-art STPG methods have been investigated: HITEC, DUST, FOG-BUSTER and MOSAIC. Shortcomings, disadvantages, but also useful techniques from these STPG methods have been described.

Based on the identified unsolved problems and previous work in STPG, new TPG techniques and processes in an ATPG system for sequential circuits have been proposed. The STPG, part of this system, applies the CTP approach and is based on the combina
tional TPGs of DAT. A new power-set based signal model has been proposed to over-
come two of the three identified problems: Incorrect bus-conflict detection and incomplete and/or inefficient search in TPG. Finally, techniques and processes have been proposed to solve the third identified problem: Illegal state space detection and over-specified identi-
fied ISes.
Chapter 5

ATPG for sequential circuits

This chapter describes the new ATPG system proposed in Chapter 4. This new system incorporates a FAN-based STPG, together with pre-processes and learning techniques to support the STPG.

An STPG is presented which uses the CTP approach, and incorporates existing and new techniques. The STPG uses the 25V signal model (however, in Chapter 6 an improved STPG method using a power-set based logic signal model will be proposed). To increase the overall performance of the STPG, various strategies are introduced. They are selected during STPG depending on properties of the circuits.

Two learning techniques are introduced: The first one is a static learning technique to identify global implications. Experimental results demonstrate that a large number of global implications is found, requiring relatively short CPU times as compared to current existing methods. The second technique tries to find signal lines with a fixed value.

A single pattern, single timeframe, single fault, explicit Fault Simulator (FS) is presented. This FS can also facilitate the selection of the next target-fault during ATPG.

Finally, additional target TPG is presented: When a test sequence has been generated for a fault, an additional target is selected in order to extend the sequence to cover the additional target. Experimental results demonstrate the effectiveness of various forms of additional target TPG, in terms of increasing the fault coverage for circuits.

This chapter is organized as follows: Section 5.1 describes the pre-processes. Section 5.2 presents the proposed STPG. Section 5.3 introduces various strategies for STPG. Section 5.4 presents the static learning techniques. Section 5.5 describes the fault simulator. Section 5.6 presents additional target STPG. Section 5.7 gives an overview of the DAT ATPG system. Finally, Section 5.8 provides a summary and conclusions.
5.1 Pre-processes

Pre-processes are performed before test generation; some are mandatory (such as controllability and observability calculations, headline determination; otherwise the STPG cannot perform or performs very poorly), others are facultative (fault collapsing, and the identification of untestable faults). The purpose of these processes is to make the task of the test generation process easier, usually improving the performance of TPG.

Fault collapsing is performed to reduce the number of faults that has to be considered in TPG (see Page 34). Headlines are determined (see Section 3.2, Page 42) to limit the number of (re)decisions and backtrace steps in TPG. In this section, three pre-processes are presented: First, to determine whether a signal line can become Z and/or U, a pre-process "can-be-Z and can-be-U calculation" is performed (Subsection 5.1.1). Second, to guide the decision-making process during STPG, controllability and observability cost estimates are required. Therefore, a pre-process to calculate these cost estimates in sequential circuits is performed (Subsection 5.1.2). Third, a pre-process for (pre-)identification of untestable faults is performed (Subsection 5.1.3), because often many faults can be identified to be untestable without explicit test generation.

5.1.1 Can-be-Z and can-be-U calculation

Knowledge on whether the signal lines can (or may) become Z or U reduces the size of the search space, and thus may limit the number of backtracks during (S)TPG, because only signal lines which are marked 'can-be-Z', or which are marked 'can-be-U' and are driven by a PPI, have three decision options (values 0, 1, and Z or values 0, 1, and U, respectively; see Section 4.1.2, Page 57), while the other signal lines have two options. It is not possible that a signal line has four decision options, because a PPI, the output of an FF, can not become Z. Similarly, a PI will never actually be driven by a U value.

A simple pre-process determines for each signal line, which values it can (may) assume, and labels the appropriate lines can-be-Z or can-be-U:

- can-be-Z: A signal line is marked can-be-Z when the line drives an element which can handle value Z (e.g., a bus element), or the line is driven by an element which can produce value Z (e.g., a switch).

- can-be-U: A signal line is marked can-be-U when the line is driven by elements which can assume value U (i.e., bus elements, PPIs, or PIs fixed at U (see Section 2.4)).

5.1.2 Controllability and observability cost calculations

To guide the decision-making during test generation, controllability and observability values are assigned to each signal line. The controllability to 0(1) of a signal line is an estimate of the cost of justifying this line to value 0(1). The observability of a signal line is an estimate of the cost of making a value change on this line visible on a PO. Similar to many other STPG methods (e.g., DUST[Gou93] and Essential [Aut91]), our
5.1. Pre-processes

![Diagram](image)

**Figure 5.1:** a) Two partial timeframes of a circuit with unjustified AND gate; b) Four partial timeframes of a circuit with unjustified AND gate.

Controllability/observability calculations are based on SCOAP [Gol79], which associates six controllability and observability values to each signal line to characterize its testability in a sequential circuit; three values to express costs in the combinational part of the circuit and three values to express the costs of the sequential behavior of the circuit. Before we propose a modification to this scheme, we will in some detail explain the SCOAP values, and show how their direct use leads to a problem.

The combinational and sequential controllability and observability estimates in general are related to the (minimal) number of required assignments to justify a certain objective:

- The **combinational controllability** to \( v \) \((v \in \{0, 1\})\) of a signal line \( i \) \((CC^v(i))\) is related to the minimum number of assignments in the circuit required to justify \( v \). This means, for example, that the \( CC^0 \) of an AND gate output is the minimum of the gate’s input \( CC^0 \)'s, and the \( CC^1 \), as the sum of the input \( CC^1 \)'s.

- The **combinational observability** of signal line \( i \) \((CO(i))\) is related to the minimum number of assignments required to make a sensitive path between line \( i \) and a (P)PO. For example, the \( CO \) of an AND gate input is calculated as the \( CO \) of the output plus the sum of \( CC^1 \) of the gate’s other inputs.

- The **sequential controllability** to \( v \) of signal line \( i \) \((SC^v(i))\) is related to the minimum number of timeframes that have to be traversed to justify value \( v \) on line \( i \).

- The **sequential observability** of signal line \( i \) \((SO(i))\) is related to the minimum number of timeframes that have to be traversed from line \( i \) to a PO, and the minimum number of timeframes that have to be traversed to control all required values to make a sensitive path between line \( i \) and this PO.

A problem in using the SCOAP values directly is that either \( SC^v \) or \( CC^v \) has to have priority (similar reasoning holds for \( SO \) and \( CO \)). This may lead to too long test sequences and/or unnecessary effort in TPG, as the following examples illustrate: Figure 5.1a shows two (partial) timeframes of a circuit. The first two values of the triple value associated with the signal lines in the circuit are the \( CC^0 \) and \( SC^0 \) (the third value will be explained further on with our proposal). The output of the AND gate has to be justified to 0; which
Table 5.1: Merged controllability for sequential circuits

can be done by assigning either line $a$ to $0$ or line $b$ to $0$. Of those two lines the one with the lowest estimated (i.e., "easiest") controllability cost has to be chosen. Normally, $SC^0$ has priority over $CC^0$ (to keep the sequence length small), hence line $a$ is chosen ($(SC^0_a = 0) < (SC^0_b = 1)$). However, assigning $0$ to line $b$ requires less effort for the TPG than assigning $0$ to line $a$, because justification of value $0$ on line $a$ requires three assignments on the PIs and justification of value $0$ on line $b$ only one assignment (see figure). Another example: Figure 5.1b shows four (partial) timeframes of a circuit. Again, the output of the AND gate has to be justified to $0$. If $CC^0$ has priority over $SC^0$, then line $b$ is chosen ($(CC^0_b = 1) < (CC^0_a = 3)$). However, assigning $0$ to line $a$ results in a (much) shorter test sequence than assigning $0$ to line $b$.

In order to solve the priority problem of the SCOAP cost values, we propose to merge the $SC$ and $CC$ values into one combined cost estimate. The Merged Controllability value $MC^v (v \in \{0, 1\})$ incorporates the combinational as well as the sequential properties of the circuit. In order to limit the length of the generated test sequences, the sequential properties have a higher weight in the $MC$ than the combinational properties. In this way, the sequential properties of the circuit have a certain degree of priority over the combinational priorities, but no absolute priority as in the case with two separate cost values. Table 5.1 shows how $MC^v$ is calculated. The $MC$ values of the signal lines are calculated by performing one or more passes over the circuit. During the first pass over the circuit, the $MC$ values of all signal lines are calculated based on the initial values on the PIs and PPIs (see block P1 in Table 5.1 and [GoI79] [vdL96]). The initial values are for PIs $MC^v = 1$ (line A in table), and for PPIs $MC^v = \infty$ (line B); i.e., initially we assume that the TPG can control the PIs and cannot control the PPIs. In the second and possibly later passes, the $MC^v$ values of those PPIs are (re)calculated for which the corresponding PPO has got a new $MC^v$ value in the previous pass: The $MC^v$ value of the corresponding PPO ($MC^v_{curr}(PPO)$) is taken and increased with a constant value ($100)^{1}$ (see line D in Table 5.1 ($MC^v_{tmp}$)). Due to this constant value, the decision-

\[1\] This value has been determined experimentally. For ISCAS’89 circuits and other comparably sized
### 5.1. Pre-processes

<table>
<thead>
<tr>
<th>First pass</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PO:</strong> ( MO = 1 )</td>
</tr>
<tr>
<td><strong>PPO:</strong> ( MO = \infty )</td>
</tr>
<tr>
<td>all other elements as described in [Gol79] and [vdL96]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Second and later passes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PO:</strong> –</td>
</tr>
<tr>
<td><strong>PPO:</strong> ( MO_{\text{tmp}} = (MO_{\text{curr}} (PPI) + 100) )</td>
</tr>
<tr>
<td>( MO_{\text{new}} = (MO_{\text{tmp}} &lt; MO_{\text{curr}}) ? MO_{\text{tmp}} : MO_{\text{curr}} )</td>
</tr>
<tr>
<td>all other elements, which drives PPOs:</td>
</tr>
<tr>
<td>( MO_{\text{tmp}}: ) as in first pass, however:</td>
</tr>
<tr>
<td>( MO_{\text{new}} = (MO_{\text{tmp}} &lt; MO_{\text{curr}}) ? MO_{\text{tmp}} : MO_{\text{curr}} )</td>
</tr>
</tbody>
</table>

**Table 5.2:** Merged observability for sequential circuits

Making process gives more priority to assignments in the current timeframe. If this newly calculated MC value \( MC^v_{\text{tmp}} \) is smaller than the current value on the PPI \( MC^v_{\text{curr}} \), it is assigned to this PPI (see line E in Table 5.1). The new PPI \( MC^v \) values are propagated in event-driven fashion through the circuit. A new \( MC^v \) value for a signal line is only assigned when it is lower than the current \( MC^v \) value (see line F).

New \( MC \) calculation passes are performed when there have been PPOs whose \( MC^v \) cost values were reduced in the last pass; i.e., the calculation process stops when \( \forall PPIs : (MC^v_{\text{PPI}} + 100) \leq MC^v_{\text{PPO}} \). Note that lines which only are driven by PIs are never recalculated in second or later passes (see line C).

In the examples of Figure 5.1, the third value of the value triplets is the \( MC^0 \). Based on these values, the backtrack process chooses line b in the circuit of Figure 5.1a and line a in the circuit of Figure 5.1b, to justify value \( \theta \) on the AND gate outputs (in this tiny example, the constant value, to be added to \( MC^v \) when passing a timeframe boundary is kept at 1 rather than 100).

An analogous reasoning holds for combinational and sequential observability: Normally, \( SO \) has priority of \( CO \), which may lead to unnecessary TPG effort (in the same way as has been explained for controllabilities, see Page 77). Therefore, we also propose to merge the \( SO \) and \( CO \) values into one combined cost estimate \( MO \). Table 5.2 shows the calculation rules for the merged observability values \( MO \).

In 3-state circuits a third controllability value is required, to steer the decision-making for value \( Z \): The controllability to \( Z, MC^Z \). Of course, for Boolean gate outputs this value is \( \infty \), as their output value cannot become \( Z \). For all 3-state elements types, \( MC^Z \) can be calculated similar to \( MC^0 \) or \( MC^1 \), by selecting the lowest-cost input value combinations. For a detailed description of the (combinational) controllability and observability calculations for 3-state elements see [vdL96].

---

circuits this constant was found to be satisfactory. However, for larger sequential circuits this value may have to be increased, because the \( MC^v \) values are higher, reducing the influence of the constant value on the \( MC^v \) values if it is not increased.
Concluding this section, two advantages can be mentioned for merged controllability and observability values: It results in a better decision making, and it requires only one rather than two controllability/observability comparisons per signal line during the backtrace- or fault propagation process. In addition, the merged values save some memory usage, because three (or four when a signal line can have value $Z$), instead of six (or eight) cost estimates per signal line are stored.

### 5.1.3 Identification of untestable faults prior to TPG

A test generator often spends a large portion of its computing time in trying to identify untestable faults. A fault is identified as untestable only when the test generator has (implicitly) enumerated the complete search space and failed to find a test for the fault. Therefore, algorithms have been developed to identify untestable faults without STPG [Agr93] [Pom94b] [Iye94] [Iye96] [Lon95]. Usually these algorithms are much cheaper in terms of computing time, compared to STPG; they can identify a large part of the untestable (including many redundant) faults of a circuit. There are five methods to perform Untestable Fault Identification (UFI) without STPG:

1. **UFI based on illegal value combinations [Iye94]**. This method is based on the concept that a fault which requires an illegal combination of values as a necessary condition for its detection is untestable. E.g., if a fault requires on a single line value 0 as well as value 1 for its detection, the fault is untestable. This method determines for each line the set of faults $S_0(S_1)$ which require value 0(1) on that line for detection; the intersection of $S_0$ and $S_1$ results in a set of untestable faults. FIRE [Iye94] (for combinational circuits), and FUNTEST [Iye94] and FIRES [Iye96] (both for sequential circuits) are based on this approach. The authors of [Iye94] and [Iye96] claim that FUNTEST and FIRES find many untestable faults which cannot be identified by an STPG (they used GENTEST [Cha91]); however, our experimental results (see Section 7.6) show that these untestable faults are easily identified by our STPG.

2. **UFI using illegal states**. This method is based on the fact that the assigned part of an illegal state is an illegal value combination (see first UFI method); i.e., faults which require the assigned values of the IS for detection are untestable. In [Lon95], FUNI has been proposed, which is based on this concept. This method identifies only "easily" identifiable untestable faults; an STPG proves these faults as untestable, as soon as the IS has been identified.

3. **UFI in static learning [Aut91]**. Static learning is a method to identify global implications; i.e., it tries to find logic value implications which cannot be determined by local reasoning. In addition, static learning tries to identify fixed values on signal lines (i.e., these lines can only assume one value, 0 or 1), using the logical inference rule of constructive dilemma (see Section 5.4.1). The SA-$v$ fault on a line is untestable, when that line is fixed at value $v$. 


5.1. Pre-processes

4. UFI based on the identification of uninitialized and unobservable signal lines. Signal lines which can not be initialized to \( v \ (v \in \{0, 1\}) \), using any input sequence, starting from an unknown initial state are called uninitialized to \( v \). The corresponding SA-\( \bar{0} \) faults are untestable.

Signal lines are unobservable when a discrepant value on that line can not be observed on a PO, in this timeframe or any future timeframe. The corresponding SA-0 and SA-1 faults are untestable.

5. UFI using combinational test generation [Agr93]. This method identifies untestable faults using a TPG for combinational circuits. In an iterative logic array of fixed length with a single fault in the last timeframe of the array, combinational test generation is performed assuming the PPIs of the first timeframe of the array to be controllable and the PPOs of the last timeframe to be observable. Faults which are proven to be untestable in this way are also untestable for STPG [Agr93].

UFI methods 1 and 2 are not considered further in this thesis; the untestable faults which are identified by these methods are also easily identified by the STPG or UFI methods 3, 4 or 5. UFI method 3 is described in detail in Section 5.4, where static learning based on implication of signal values is explained. In the next paragraph a pre-process is described to identify uninitialized and unobservable signal lines; this information is used in UFI method 4. UFI method 5 is applied within our ATPG system; however, only for an array of length 1. The identified untestable faults are Combinationally Redundant Faults (CRFs; see Section 2.5.3).

Uninitializeable and unobservable signal lines. The identification of uninitialized signal lines (i.e., lines which can not be initialized to 0 and/or 1, using any input sequence, starting from the unknown state) and unobservable signal lines (i.e., lines which value cannot be made observable on a PO) before STPG, simplifies the test generation process significantly: Many faults which require those lines to be initializeable to 0 or 1, or to be observable, can be marked untestable immediately (UFI method 4). In addition, STPG can start backtracking when during test generation values on lines are requested which are uninitialized, and/or the fault has to be propagated to POs over signal lines which are unobservable.

The pre-process, called FUN (Find UNinitializeable and UNobservable lines), can be divided into two parts:

1. It calculates for each signal line the initializability to 0(1); i.e., it calculates whether a 0(1) on a signal line can be justified with an input sequence starting from an unknown initial state.

2. It calculates for each signal line the observability; i.e., it calculates if a discrepant value on a signal line can be observed on a PO, in this timeframe or any future timeframe.
Figure 5.2: a) Sequential circuit  b) Two timeframes of sequential circuit of Figure a. FF is not initializeable to 1

The identification of the initializeable signal lines is straightforward: All signal lines initially are marked uninitializeable to 0 and 1. Then all PIs are marked initializeable to 0 (denoted with the Boolean flag init0) and initializeable to 1 (denoted with the Boolean flag init1). A forward pass is performed over the circuit, starting from the PIs/PPIs, terminating at the POs/PPOs. An output of an element is marked init0(init1), when there is an input combination possible which justifies the output of the element to value 0(1). For example, the output of an AND gate is init1 if all inputs are init1; the output of an AND gate is init0 if there is at least one input init0. Multiple passes over the circuit are performed as long as there are PPOs marked init0(init1) while their corresponding PPI is not marked init0(init1); i.e., forward passes continue until all PPO/PPI combinations are marked equally. Figure 5.2a shows an example circuit to demonstrate the process of identification of uninitializeable lines; Figure 5.2b shows the same circuit over two time-frames. Initially all lines of the circuit are marked uninitializeable to 0 and 1; then the PI is marked init0/init1 (see timeframe 1 of Figure 5.2b). The output of the AND, which is a PPO is marked init0. Its corresponding PPI (still) is not marked init0; therefore, the procedure starts a second pass in timeframe 2, and marks the PPI init0. The output of the AND is already marked init0 and the procedure stops. It has found that the FF in Figure 5.2a cannot be initialized to 1.

For all signal lines which are marked uninitializeable to \( v (v \in \{0, 1\}) \), the corresponding SA-\( \bar{v} \) fault is flagged untestable and removed from the fault-list; these faults cannot be activated. This process and the controllability cost estimate calculations (see Section 5.1.2) are combined into one process to save computing time.

The identification of unobservable signal lines is performed in combination with the observability cost estimate calculations, as described in Section 5.1.2. Initially, all POs are marked observable and all PPOs are marked unobservable. A backward pass is performed over the circuit, starting from the POs/PPOs, terminating at the PIs/PPIs. For each element input there are three possibilities:

1. An input of an element is marked observable when the output of that element is observable and the remaining inputs are initializeable to the non-dominating value. For example, an input of an AND gate is observable when its output is observable and the remaining inputs are init1.

\(^2\)i.e., a signal line is only visited when its initializeability has been changed.
2. An input of an element is marked *unobservable* when the output of that element is marked unobservable. For all signal lines which are marked unobservable, the corresponding SA-0 and SA-1 faults are flagged untestable and removed from the fault-list; the fault-effects of these faults cannot be propagated to a PO.

3. An input of an element is marked *potentially unobservable* when the output of that element is observable, but one or more of the remaining inputs of the element can not be initialized to the non-dominating value of that element. Figure 5.3a shows a part of a circuit consisting of an FF and an AND gate. The FF is not initializeable to 1. Therefore, in the fault-free circuit, input a of the AND gate can not be observed on output PO. However, in the *faulty* case there is a chance that the fault may initialize the FF to 1 and thus making input a observable on the output. Therefore input a is marked *potentially unobservable*.

Figure 5.3b shows a circuit with a SA-1 fault on line f. This fault is testable, because the fault-effect can be observed on the PO via input a of the AND gate. The *fault* initializes FF1 to 1, which is required for fault observation at the PO. Note that FF1 is not initializeable to 1 in the fault-free case. In the case of the SA-1 fault on line f, FF1 does not have to be initialized in the fault-free case, because value 0 is on input a, which dominates the value on input b of the AND gate. Note that the SA-0 fault on line f is not testable; in that case initialization to 1 of FF1 is required in the fault-free circuit.

Signal lines which are marked potentially unobservable can be (temporarily) marked unobservable during STPG for a target fault, when it is proven that this fault does not initialize FFs which are uninitializeable in the fault-free case (see Section 5.2.3). No STPG effort has to be performed to propagate the fault to these unobservable lines. The lines are restored to being marked potentially unobservable when the test generation process for the fault has finished.

### 5.2 Deterministic test generation

In this section, the STPG of the ATPG system DAT is presented. The STPG uses the iterative array model. Test generation is performed using the CTP approach over time-
frames, and inside timeframes FAN-algorithm [Fuj83] [vdL94b] based TPG is performed. The STPG is described in a top-down manner: In Subsection 5.2.1, the CTP approach of the STPG is described; three STPG phases are defined. In Subsection 5.2.2, a description is given of the STPG process within one timeframe. Thereafter, in Subsections 5.2.3 through 5.2.8, six aspects of the STPG are explained in detail because the FAN algorithm has been modified significantly and/or new functionality/techniques have been added. These six aspects are:

1. Fault dependent cost estimates and the concept of local (inside one timeframe) and complete fault-cones (over timeframes).

2. Fault propagation and Dynamic Dominator (DD) determination.

3. Multiple backtrace using objectives with six counters, to represent value requests of signal lines in the fault-free as well as the faulty circuit.

4. Decisions with three value options on circuit lines.

5. Internal restrictions: The State Restrictor, which constrains the state space in the fault-free and the faulty circuit; and internal restrictions imposed by buses and other (3-state) elements which can cause bus-conflicts.


5.2.1 Three phases during STPG: Activation, propagation and justification

The STPG uses the CTP approach, which usually consists of two phases: The first phase is the FTP phase in which the fault is activated and propagated to a PO. The second phase is the RTP phase for justification of the state requirements determined in the first phase. However, we have split the first phase into two phases because fault activation is only performed in the first timeframe; fault propagation is performed in the first and later timeframes. Hence, the STPG is divided into three phases (see Figure 5.4):

1. The fault activation phase (phase 1): Imply the fault $F$ in the circuit in timeframe 1, activate the fault, and propagate the fault to one or more (P)POs by exhaustive implication of mandatory assignments and decisions on PIs or PPIs, and backtracking.
5.2. Deterministic test generation

when necessary. We treat this phase as a separate phase because (only) in timeframe 1 the fault is activated as well as propagated.

2. The forward propagation phase (phase 2): As long as the fault is not propagated to at least one PO, a new timeframe (e.g., timeframes 2 and 3 in Figure 5.4) is created and the values (including one or more discrepant values) of the (assigned) PPOs of the previous timeframe are implied on the corresponding PPIs of this timeframe. The fault-effects (i.e., discrepant values) on the PPIs are propagated to at least one PPO or PO (in a similar way as described for phase 1).

3. The backward justification phase (phase 3): Justify backward in time all requirements on the PPIs until a state is reached which covers the last generated state of the previously generated test sequence. In case it is the first generated test sequence, always the all-unknown state has to be reached. E.g., if all objectives are satisfied in timeframe 4 (see Figure 5.4) (i.e., the requirements of timeframe 1 are fulfilled), and timeframe 4 has specified values on the PPIs, which are not covered by the last generated state of the previously generated test sequence, STPG creates timeframe 5 and implies all fixed known values of the PPIs of timeframe 4 on the corresponding PPOs of timeframe 5. These implications result in a set of objectives which have to be justified.

The result of STPG for a single SAF $F$ can be one of the following:

- The fault $F$ is tested. In that case STPG has succeeded.
- The fault $F$ is untestable. In that case STPG has proven that the fault is untestable.
- The fault $F$ is aborted. In that case STPG has been aborted, because one or more limits have been exceeded. STPG is constrained by a time limit in seconds, a maximum test sequence length, and a global backtrack limit for all timeframes, or a backtrack limit per timeframe together with a limit on the number of created timeframes.

5.2.2 Overview of the STPG algorithm

In the previous section a description has been presented of the STPG process over the timeframes. In this subsection the STPG process inside a single timeframe is explained. Figure 5.5 shows the flow diagram of the STPG process. The test generation process starts with a single fault (in the top of the figure). It terminates by reporting the test result (fault is tested or untestable), whereby a test sequence is delivered if the fault was tested. For simplicity, limits are not considered in this diagram; i.e., the test generation process cannot be aborted. Hereafter, a short description is given of each box/ellipse (labeled with $a \ldots l$) in the flow diagram.

**Box α.** STPG starts with a calculation of the fault dependent cost measures (see Subsection 5.2.3).
**Ellipse b.** The first timeframe is created (phase 1, the fault activation phase starts). In that timeframe, the faulty value (value 0 for SA-0; value 1 for SA-1) is implied, the fault is activated (imply value 1 for SA-0; value 0 for SA-1 in the fault-free circuit), resulting in a list of unjustified values, and the D-front is determined. If conflicts are detected during implication and/or D-front determination (i.e., a failure is detected; see Figure 5.5), then the fault is untestable. Otherwise the process continues with Box c.

**Box c.** Flag *make_new_objs* is set. This denotes that before the backtrace procedure starts, the lists of Initial Objectives (IO list), Current Objectives (CO list), Fan Objectives (FanO list), and Head Objectives (HO list) are cleared. A new IO list is created from the current unjustified lines. In addition, if the fault-effect is not yet propagated to a PO, then one objective is added in the IO list, to propagate the current D-front one level closer to the PO/PPOs (see also FAN algorithm in Section 3.2, Page 41).

**Ellipse d.** STPG checks if fault propagation is completed; i.e., the D-front has reached one or more POs or PPOs. If only PPOs have been reached, an additional require-
Figure 5.6: 3 timeframes during the fault propagation phase; current PPO state equals the PPO state of timeframe 1, hence STPG continues fault propagation in timeframe 3

Figure 5.7: Depending on the current phase in STPG, the next timeframe is selected backward or forward in time

...ment has to be fulfilled: The current PPO state may not be equal to any previous generated PPO state for this fault. Figure 5.6 illustrates the case that the current PPO state equals a previous PPO state, namely timeframe 1; then oscillation has been detected. In that case, fault propagation is not finished and another D-front element has to be propagated to the POs/PPOs.

Ellipse e. When fault propagation has not completed, or there are still objectives in the current timeframe which have not been justified, Ellipse f is performed (backtrace procedure); otherwise the process performs Ellipse i (bus-padding procedure).

Ellipse f. A Final Objective (FO) is determined by the Multiple or Single Backtrace (MB/SB) procedure (see Page 41).

Ellipse g. The FO from the backtrace procedure (Ellipse f) leads to a decision by the STPG, which is stored on the decision stack and implied in the circuit. After the decision has been implied, the STPG process jumps back to Ellipse d, where the fault propagation check is performed.

Ellipse h. When fault propagation has not completed, Dynamic Dominator (DD) determination is performed. This has been described in Section 3.2.2, Page 45 for CTPG. However in Subsection 5.2.4, modifications are proposed for the STPG.

Ellipse i. If fault propagation has completed and all objectives in the current timeframe have been justified (Ellipse e) then, for circuits with 3-state elements, bus-padding (see Page 45) has to be performed.

Ellipse j. The STPG is ended when the fault is detected; i.e., the fault is activated and propagated to a PO, and all (PPI) states are justified. Otherwise, the STPG process continues to Ellipse k, where the next timeframe is selected and initialized.
Figure 5.8: Drivability costs of value $D$ on the output of a NAND gate

**Ellipse $k$.** The next timeframe is selected and initialized. In which direction the next timeframe is selected depends on the current phase. Figure 5.7 visualizes the next-timeframe procedure: In phase 1 (fault activation) and 2 (fault propagation) the next timeframe is selected forward in time. In phase 3 (state justification) the next timeframe is selected backward in time.

After the next timeframe has been selected, the timeframe is initialized by copying the assigned PPO values of the previous timeframe to the corresponding PPIs of the new timeframe (phase 2), or by copying the assigned PPI values to the corresponding PPOs of the new timeframe (phase 3). The assignments are exhaustively implied, resulting in new unjustified lines. In addition, the D-front is determined when STPG is in phase 2.

**Ellipse $l$.** During the execution of the procedures of Ellipses $f$, $g$, $h$, $i$, and $k$, conflicts can be detected. E.g., assignments of values on signal lines which cannot become that value, or an empty D-front. In this case, STPG has to perform the backtrack procedure to remake and/or discard previous decisions. If the decision stack becomes empty the fault is untestable.

### 5.2.3 Fault dependent cost estimates

In this subsection, fault dependent cost estimates are described: The controllability values of the faulty circuit (fault-dependent controllability), and the drivability [Che88] values. Also, the fault-cone concept is described. These values and the fault-cone are determined for each target fault before test generation is performed for that fault, and are required to optimize the heuristic steering process in the STPG. The purpose and meaning of controllability values have been described in Subsection 5.1.2.

The drivability values have been introduced in the BACK algorithm [Che88]. They denote the estimated costs of getting a discrepant value on a signal line; i.e., the costs of having that line be sensitive for the fault. Figure 5.8 shows a NAND gate. The minimal costs to get value $D$ on the output of the NAND is sum of the minimum of the costs of assigning value 1 or $\bar{D}$ to all inputs. In this summation at least one cost value of $\bar{D}$ has to be incorporated.

The fault-cone is defined as the collection of all signal lines which can be influenced by the fault. This influence extends over timeframe boundaries. Figure 5.9a shows the fault-cone for a fault $F$ over three timeframes. All signal lines within the fault-cone
have fault dependent cost estimates. The fault dependent controllability and drivability values of the signal lines outside the fault-cone are infinite; i.e., fault dependent values on these signal lines can not be obtained. The complete fault-cone is defined as the set of lines which can be influenced by the fault over timeframes. For example, Figure 5.9b shows the complete fault-cone of the fault-cone over timeframes 1, 2 and 3, as shown in Figure 5.9a. Figure 5.9c shows the local fault-cone of timeframe 1. The local fault-cone is the fault-cone within a timeframe, which is always a subset of the complete fault-cone.

Why are fault dependent cost values desired in STPG? In combinational circuit TPG (CTPG) only fault independent cost values are defined. The following reasons are given:

- The objectives which are handled during the backtrace procedure (current objectives, and fanout objectives; see Page 41) are always fault independent in the case of CTPG. Hence, only fault independent controllability values are required. The headlines, on which the decisions, resulting from the backtrace procedure are assigned, are always fault independent.

In a sequential circuit, the headlines can be fault dependent: A number of PPIs can be influenced by the fault in previous timeframes (see Figure 5.9a). To avoid overspecified PPIs, the backtrace procedure has to know in which circuit (fault-free or faulty) an objective has to be satisfied. So, STPG makes decisions in the fault-free and the faulty circuits.

- The initial objectives for the backtrace procedure result from unjustified lines. In a combinational circuit unjustified lines are always fault independent. In sequential circuits the unjustified lines can be fault dependent. When a signal line is unjustified and can be influenced by PPIs, the STPG has to know in which circuit the value on the unjustified lines is requested (otherwise, over-specified PPIs can occur).

- In a sequential circuit the percentage of fault dependent signal lines is usually much larger than in a combinational circuit. A signal line in a combinational circuit can only be sensitive for the fault site. In a sequential circuit a signal line can be sensitive for the fault in the current and/or previous timeframes.

- The fault in the circuit modifies the circuit. Especially in sequential circuits (the fault-effect is propagated over timeframe boundaries), the controllability costs of
Figure 5.10: To propagate 1/0 on line e through gate f, value X/1 is requested on line d.

Figure 5.11: a) X/1 objective results in X/0 on line d in fault-free circuit; b) 0/1 objective results in 1/0 on line a and 1/X objectives on the other input lines.

signal lines in the fault-free and faulty circuit can be very different, resulting in differences in choices during the backtrace. E.g., Figure 5.10 shows two timeframes of a circuit during the STPG process. To propagate the fault-effect on line e in timeframe 2 through gate g, value X/1 is requested on line d. The backtrace procedure will choose line c to justify line d when it uses the controllability value (MC^1) of the fault-free circuit. However, due to the fault, the controllability value of line b has been decreased to 100. Therefore, to justify line d, line b will be a better choice.

Therefore, in order to improve the decision-making process of the backtrace procedure, controllability values are calculated for the faulty circuit.

The application of drivability values in the STPG can be motivated by similar reasons as for the fault dependent controllabilities\(^3\): Figure 5.11 shows the usefulness of drivability values, compared to fault dependent controllability values. The figure shows that the backtrace of an objective with dominant value in the faulty circuit (X/1 on the output of the NAND) can result in a different input objective, then in the case where the objective is a discrepant value (0/1 on the output of the NAND). This difference of choice by the backtrace is due to two reasons: First, to justify the objective of Figure 5.11a also an input objective in the fault-free circuit is allowed. Second, an objective requesting a discrepant value, in fact consists of two objectives: One objective in the fault-free, and one in the

\(^3\)However, historically [Mar78] drivability values have been introduced/applied to guide the STPG in selecting a potential fault-sensitive path (Topological path, see Section 3.3.2).
faulty circuit. Both (different) objectives have to be justified, hence, input objectives have to be created in the fault-free and faulty circuit, and among these objectives, an objective with discrepant value is required; i.e., the “best” input objective to be selected, to justify the discrepant output value, is based on the drivability cost estimates, which actually is the combined cost estimate of the controllability cost estimates of the fault-free and faulty circuit. Hence, input $a$ in Figure 5.11b is the “best” input to justify the output value.

The fault dependent controllabilities ($FC^0$, $FC^1$), and drivabilities ($DV^D$, $DV^D$) are calculated at the beginning of STPG for a fault (Box $a$ in Figure 5.5). Together with these calculations, the signal lines in the fault-cone are identified. Furthermore, signal lines which are marked potentially unobservable (see Section 5.1.3, Page 81), are (temporarily) marked unobservable, when the fault cannot make them observable. All signal lines which are marked uninitialize to $0$ and/or $1$ in the fault-free circuit are also uninitialize in the faulty circuit, except if a SA-0(1) fault makes an uninitialize to $0(1)$ fault site initialize, in which case also other signals may become initialize. In that case, the uninitialize-information of the signal lines may not be used in the faulty circuit.

5.2.4 Fault propagation

In this subsection, fault propagation to a PO is discussed. The concept of Dynamic Dominators (DD) is extended for sequential circuits. Also three D-front flags are introduced to denote the status of fault propagation.

As has been described in Subsection 5.2.2, in the fault activation and the propagation phase, the D-front is determined; it consists of all elements in the current timeframe that have a discrepant input value, and a non-discrepant, but (still) unspecified output value. DD determination is performed using this D-front. A description of the DD procedure for combinational circuits has been given in Section 3.2.2, Page 45, which is a summary of Section 4.4.7 in [vdL96]. For STPG, the DD procedure requires a modification concerning the off-path signal lines. Figure 5.12 shows a circuit with a SA-0 fault on input line $c$ of the AND gate. The element inputs marked by a star (*) are fault sensitive (i.e., they are in the (complete) fault-cone; lines $a$, $f$, etc., possibly can be reached by fault effects from previous timeframes). The D-front consists of element $A$. The DD procedure is applied to this D-front element: It assigns the non-dominant value to the off-path signal lines of all elements which must be traversed (i.e., are dominators of the D-front element(s)) when propagating the fault to a PPO/PO. However, these off-path signal lines can be fault dependent and are part of the complete fault-cone, but not of the local fault-cone. To propagate a discrepant value through an element, a non-dominant value is required on all off-path, fault-insensitive signal lines, and on either the fault-free or faulty part of the fault-sensitive (fs) signal lines. For example: Consider the discrepant value on line $c$ in Figure 5.12. To propagate this value through element $A$, value $i$ is required on the fault-insensitive signal line $b$ and value $1/X$ on the fs-signal line $a$. Which value is assigned to the off-path fs-signal lines depends on the parity of the on-path signal line(s) (bold lines in the figure, and part of the local fault-cone). An (on-path) signal line has a positive (negative) parity ($+(-)$) when the (expected) discrepant value on this line is $1/0$ $(0/1)$. Hence, line $c$ has a positive parity, line $e$ has a negative parity, and line $g$ has a positive parity.
Figure 5.12: To determine the mandatory off-path signal values for the dynamic dominators procedure, the fault-sensitivity parities of the on-path signal lines have to be determined.

<table>
<thead>
<tr>
<th>Parity</th>
<th>(N)AND</th>
<th>(N)OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>1/X</td>
<td>X/0</td>
</tr>
<tr>
<td>-</td>
<td>X/1</td>
<td>0/X</td>
</tr>
</tbody>
</table>

Table 5.3: Mandatory assignments to the off-path fault sensitive input lines, based on the on-path parity and the element type

Based on the parity and the (Boolean) element type of a dominator, a value is assigned to the off-path fs-signal lines. Table 5.3 shows these values. Hence, value 1/X is required on line a and value X/1 is required on line f. If an element has multiple on-path inputs (such as element B in Figure 5.12) then, only when all on-path lines have equal parity, Table 5.3 can be applied to the off-path fs-lines; otherwise no value can be assigned to the off-path fs lines. E.g., although the fault has to be propagated through element B, no value can be assigned to line o, because lines n and p have different parities. Note that, as soon as the parity of a signal line is unknown (e.g., line q in the figure), it remains unknown for all on-path signal lines after that signal line.

There is no parity value required for the input signal lines of 3-state elements: The non-dominating off-path signal values on the input lines of 3-state (dominator) elements are the same in the fault-free as well as the faulty circuit. Figure 5.13 shows an example: All off-path signal lines have to become value Z, to propagate the 1/0 value on line b to

Figure 5.13: a) Non-dominant value (Z) on off-path signal lines, to propagate fault-effect (on line b), are fault independent (no parity required)
5.2. Deterministic test generation

Figure 5.14: a) Blocked fault sensitive path (bold line) is detected by the X-path-check procedure; b) Based on the parity information of line c, the X-path-check procedure detects a blocked path

line d, whether the off-path lines are inside, or outside the fault-cone.

Using the parity information, also blocked (fault sensitive) paths can be detected. These are paths over which the fault cannot be propagated (anymore) to a PPO/PO. Blocked paths are often detected using the X-path-check procedure (see Page 41). For instance, the X-path-check detects the blocked fault sensitive path (bold line) as shown in Figure 5.14a, because it notes that output d is already set to value 1. However, it does not detect the blocked path in Figure 5.14b. Using the parity information, the blocked path, as shown in Figure 5.14b, will be detected: Line c has a negative parity, meaning that if a discrepancy reaches line c, it will be 0/1. Therefore, the off-path fs-signal lines have to become value X/1. This value conflicts with the X/0 value on line a, hence, the fault sensitive path is blocked.

Three global flags are defined to register the status of fault propagation during STPG. Based on these flags, STPG determines if fault propagation is still useful in the current timeframe.

1. PReached. This flag is set when one or more POs have been assigned a discrepant value; i.e., the fault-effect has been propagated to PO. The STPG stops with the fault propagation phase, when this flag is set.

2. PUnreachable. This flag is set when, starting from the current D-front, no PO can be reached anymore within the current timeframe.

3. PPReached. This flag is set when one or more PPOs have been assigned a discrepant value.

If PPReached is set, and PUnreachable is not set, and the D-front is not empty, then STPG will continue with fault propagation in the current timeframe, because a PO still can be reached in the current timeframe.

If both PPReached and PUnreachable are set, and the current PPO state is not equal to a PPO state of a previous timeframe (see Page 86), then STPG stops with fault propagation in the current timeframe, and continues fault propagation in the next timeframe (otherwise STPG continues with fault propagation and tries to propagate another D-front element to a (P)PO).
The value of these three flags are updated after the D-front has been updated. Updating the D-front is required when assignments have been performed in the local fault-cone. The DD procedure is performed after the D-front has been (re)initialized (e.g., when a new timeframe has been created or after backtracking), and after a D-front update.

5.2.5 Backtrace with objectives having six counters

This subsection describes the modifications required for the backtrace procedure. During backtrace, three lists of objectives are maintained: The list of Current Objectives (CO), the list of Fanout Objectives (FanO), and the list of Head Objectives (HO) (see Page 41). In CTPG, the objectives in the lists each consist of a signal line and three counters, which register the requested values on this signal line: \( \{l, 0req, 1req, Zreq\} \). As described in Subsection 5.2.3, the objectives in CTPG are always fault independent. Therefore, three counters are sufficient to describe the request(s) of a signal line. In STPG there can be separate objectives in the fault-free and faulty circuit. Objectives in the faulty circuit can occur on signal lines inside the complete fault-cone, but outside the local fault-cone. E.g., mandatory assignments to off-path input lines, determined during the DD procedure, can be in the faulty circuit; these signal lines are outside the local fault-cone. Hence, there are objectives (with three counters) in the fault-free and faulty circuit separately. These objectives are combined into one objective, resulting in objectives with six counters: \( \{l, g0req, g1req, gZreq, f0req, f1req, fZreq\} \). E.g., the assigned value to line \( f \) (X/1) in Figure 5.12, can result in the objective: \( \{f, 0, 0, 0, 0, 1, 0\} \), specifying a single request for a 1 in the faulty circuit. Objective \( \{a, 1, 2, 0, 0, 1, 0\} \) for line a, specifies the requests of a 0 and 1 (2 times) in the fault-free case and a 1 in the faulty case simultaneously. The counters of the fault-free and faulty part are equal for objectives of signal lines outside the fault-cone. The backtrace procedure determines final objectives (FOs) based on the objectives in the CO, FanO and HO list, resulting in decisions in the fault-free and/or faulty circuit. The decisions in the faulty circuit are always on PPIs (or on fanout-stems, which are influenced by these PPIs). These decisions have to be justified in the previous timeframes (phase 3; state justification).

The three counters for the fault-free/faulty part of the objectives corresponds to the specified values 0, 1 and Z of the 5UV signal model. However, value \( U \) is also a specified value. This value can occur due to PIs that are fixed to \( U \), to 3-state elements, or to re-made decisions on PPIs (see Section 2.4). Although value \( U \) is a specified value, it has no corresponding counter in the objectives, because it will never be explicitly requested. Signal lines with value \( U \) are always justified.

5.2.6 Three-valued decisions and decision stacks

This subsection describes modifications related to the decision-making process in STPG. According to Section 4.1.2, certain signal lines have three decision options, rather than the usual two: Signal lines which drive 3-state elements (are allowed to become \( Z \), in addition to the values 0 and 1), and signal lines driven by PPIs (are allowed to become \( U \)), have three decision options. A three-bit flag registers which decision options have
5.2. Deterministic test generation

Figure 5.15: Decision stacks of four timeframes. The arrows express the decision-making flow, starting in timeframe 1 (labeled with 'start')

already been tried on these signal lines. On fault-dependent PPIs (i.e., they are in the fault-cone) a decision in the fault-free as well as in the faulty circuit is possible; each circuit part has three decision options. Hence, on a fault-dependent PPI nine \((3 \times 3)\) different decision options are possible. A decision, which assigns the fault-free as well as the faulty part of a value to a (fault-dependent) PPI, is always split into two decisions: One decision for the fault-free, and one for the faulty circuit. E.g., if STPG decides on a fault-dependent PPI value 1/1 then two decisions are sequentially made: 1/X and X/1. Each of these two decisions has two alternatives. The size of the search space is thus in worst case:

\[
\text{Size searchspace} = 2^n \cdot 3^r \cdot 3^{ppi} \cdot 9^{fppi}
\]

where \(n\) is the number of Boolean PIs, \(r\) the number of PIs driving 3-state elements, \(ppi\) the number of fault-independent PPIs, and \(fppi\) the number of fault-dependent PPIs.

Each timeframe has a decision stack associated to it. Figure 5.15 shows the decision stacks of four timeframes. The arrows denote the order of the decision-making: first, decisions have been made in timeframe 1 in the fault activation phase (phase 1); second, decisions have been made in timeframe 2 in the fault propagation phase (phase 2), etc. The backtrack order is the reverse of the decision-making order. E.g., in timeframe 4, backtracking continues until the stack is exhausted. Then the backtrack procedure moves to timeframe 1, and backtracks until no alternative decision options are available anymore for phase 3 in that timeframe. Then the procedure moves to timeframe 2, etc.

5.2.7 Internal restrictions

The circuit is constrained by itself in two ways (called restrictions):

1. By the state transition diagram (STD) of the circuit. The STD imposes a relation between the states. Certain states can only be reached from certain other states. Hence, the (fault-free and faulty) circuit is constrained to the reachable states starting from the initial states (see also Section 2.2.3). These are called the legal states.

2. By potentially conflicting elements in the circuit, such as 3-state buses [vdL96]. As described in Section 2.1, many 3-state elements are constrained to certain input values; i.e., there are input value combinations which are not allowed to be applied
Figure 5.16: The SR contains five lists. The arrows denote the data flow between the five lists

to the inputs of such an element. Note that these constraints apply only to the fault-free circuit.

This subsection describes these restrictions in relation to STPG. The STPG has to respect the restrictions imposed by the circuit; it is not allowed to generate vectors, which (potentially) violate the restrictions. Therefore, three techniques are introduced: First, the State Restrictor (SR) which constrains the STPG to the (still) legal states (Subsection 5.2.7.1). Second, the identification of temporary illegal states. These states are illegal within one timeframe and are sometimes detected during STPG in the backward justification phase (phase 3) (Subsection 5.2.7.2). Third, bus-padding per timeframe, to assure that the generated sequence of vectors can not cause bus-conflicts (Subsection 5.2.7.3).

5.2.7.1 State restrictor

During STPG, the SR constrains the state (search) space. The SR consists of five IS lists (see Figure 5.16): Three lists keep illegal PPI states, and two lists keep illegal PPO states. The lists are required to prevent oscillation, and to avoid entering illegal parts of the search space. These lists are described in the next two paragraphs.

Illegal PPI state lists. There are three IS lists which contain ISes of the PPIs:

1. Used State List (USL): During the backward justification phase (phase 3), each state, which has to be justified, is stored in this list (arrow 1 in Figure 5.16); i.e., the PPI states of timeframe 1 (generated in the fault-activation phase), and all PPI states of the timeframes before timeframe 1 (generated in the phase 3), are stored in the USL. The states generated after timeframe 1 (i.e., in phase 2, the fault propagation phase) do not have to be stored in this list, because they are stored in the PPO state lists (see next paragraph). E.g., Figure 5.17a shows four timeframes (1 . . . 4); when all objectives of timeframe 1 have been fulfilled, STPG stores the current PPI state (000) in the USL (see Step 1 in Figure 5.17b), and continues with timeframe 2 to justify PPI state1 (the PPI state of timeframe 1).

When during the backward justification phase, a state on the PPIs is covered by one of the states in the USL, a loop is detected and STPG has to backtrack. E.g.,
to justify $PPIstate3$ (see Figure 5.17a), $PPIstate4$ has to be 000. But this state equals (or is covered by) $PPIstate1$, which has been stored in the USL (Step 1 shows the contents of the USL); i.e., a loop is detected and STPG starts to backtrack to prevent oscillation.

If a state cannot be justified, the STPG moves back to the previous timeframe, removes this state from the USL, and stores it in one of the next two lists. E.g., assume that the only way to justify $PPIstate3a$ is 000; however, this state loops with $PPIstate1$. $PPIstate3a$ is removed from the USL, and $PPIstate2$ has to be justified in an other way (Step 2 in the figure).

2. Fault-dependent Illegal State List (FISL): The states in this list are illegal because they cannot be justified, but this holds only for the current fault-under-test. Two PPI state types are stored in the FISL:

- *Directly* fault-dependent PPI states. These (illegal) states have entries with discrepant values (1/0 or 0/1, but also X/0, 1/X, etc. are possible).

- *Indirectly* fault-dependent PPI states. These can only be justified with a state currently in the USL. E.g., the reason that $PPIstate3a$, which is stored in the FISL, cannot be justified is that it requires state $PPIstate1$, which is currently in the USL.

The states stored in the FISL always come from the USL (arrow 3 in Figure 5.16). Figure 5.17a shows that an alternative PPI state can justify $PPIstate2$ ($PPIstate3b$: 100). However, this state requires state 001, which loops with $PPIstate2$. A third alternative state to justify $PPIstate2$, does not exist, and
thus also (the unjustifiable) \( PPIstate2 \) is moved from the USL to the FISL (Step 4 in Figure 5.17b).

3. **Global Illegal State List (GIS list)**: The states in this list are global illegal; i.e., these states can never be justified, independent of the fault. For these states no initialization sequence exist.

   After Step 4, STPG is back in timeframe 2. Assume that no alternative state can justify \( PPIstate1 \), then \( PPIstate1 \) is globally illegal because it does not require a state currently in the USL; i.e., \( PPIstate1 \) is illegal, independent of (PPI) states generated for the fault. All PPI states stored in the FISL, after that \( PPIstate1 \) was stored in the USL (Step 1) are also globally illegal, because these states require \( PPIstate1 \) to be justified, and this state is globally illegal. Step 5 shows this situation. So, the states stored in the GIS list can come from the USL directly (arrow 4 in Figure 5.16), or from the FISL (arrow 5).

**Illegal PPO state lists.** There are two illegal state lists which keep illegal states of the PPOs:

1. **PPO Used State List (OUSL)**: During the forward propagation phase (phase 1 and 2), each PPO state is stored in this list, before moving to the next timeframe (arrow 2 in Figure 5.16). The STPG uses the OUSL to prevent oscillation in the fault propagation phase.

2. **PPO Illegal State List (OISL)**: The states in this list cannot propagate the fault to a PO. Figure 5.18 shows an example. Assume that starting from \( PPIstate2 \), STPG fails to propagate the fault to any PO. Then STPG returns to timeframe 1 (to remake the top decision on the decision stack in that timeframe), and moves \( PPOstate1 \) from the OUSL to the OISL (arrow 6, in Figure 5.16).

Fault propagation within one timeframe completes when a PO has a discrepant value (the forward propagation phase stops, and the STPG starts with phase 3, the backward justification phase), or when a PPO has a discrepant value and the current PPO state is not equal to any of the states in the OUSL and OISL (see also Page 86). If the PPO state is equal to a state in the OUSL or OISL, STPG has to continue to propagate more discrepant values to the PPOs, until the current PPO state differs with all states in the two
5.2. Deterministic test generation

Figure 5.19: Decision stack of timeframes 2 and 3. PPIstate3 is unjustifiable, however this state cannot be stored as (G)IS, because the decisions of timeframe 2, made in the fault propagation phase (phase 2), may have caused that PPIstate3 cannot be justified. This state is temporary illegal for timeframe 3 in phase 3.

lists, in order to prevent oscillation. If this is not possible; i.e., no D-front elements are available anymore, backtracking has to be performed. Both PPO IS lists are only used in the forward propagation phase.

Note 1: Considering the implementation of the PPO IS lists, only one PPO list is necessary: The current PPO state has to be not equal to any of the PPO states stored in both lists. However, for symmetry with the PPI lists and for clarity, we introduced two PPO IS lists.

Note 2: Assume that, starting from PPIstate2, as defined in Figure 5.18, STPG can propagate a discrepancy to a PO. However, for some other reason conflicts are detected (for example, the PPIstate1 could not be justified), and as a result, the backtracking procedure has to return to timeframe 1, phase 1. Then, PPOstate1 is removed from the OUSL and discarded, instead of stored in the OISL. Because, the reason for backtracking was not the unpropagatable D-front of PPOstate1 (but in this case an illegal PPIstate1 and thus stored in one of the lists of the SR)); starting from PPOstate1, STPG can reach a PO, so it is allowed for the STPG to generate this state again.

Backtracking is also performed when the current PPO state is covered by a state in the GIS list, because this PPO state is equal to the PPI state of the next timeframe. Hence, no PPI or PPO state is allowed to be covered by any state in GIS list.

5.2.7.2 Temporary illegal states

The concept of temporary illegal states is explained using the example shown in Figure 5.19. Assume that the decision stack of timeframe 2 in phase 3 (backward state justification) is exhausted. So, STPG moves to the previously handled timeframe, timeframe 3. PPIstate3 can not be justified (otherwise the backtracking procedure had not moved from timeframe 2 to timeframe 3). However, the state is not stored in the state restrictor as being illegal, because PPIstate3 may be justifiable without the decisions made in timeframe 2 for phase 2 (the fault propagation phase); these decisions constrained the search space, when justifying PPIstate3 in timeframe 2. Therefore, PPIstate3 is called a temporary illegal state.

Backtracking on decisions of phase 3 (or 2) in timeframe 3 must result in a differ-
ent $PPI_{state3}$, to have a chance to be justifiable; i.e., the new $PPI_{state3}$ may not be covered by the temporary illegal state(s) of this timeframe. After backtracking in timeframe 3, STPG continues in timeframe 3 until all requirements are (again) satisfied. This implies also that the temporary illegal state(s) of this timeframe do(es) not cover the PPI state. The temporary illegal state(s) are discarded when, due to backtracking, timeframe 3 is destroyed, and the STPG (in phase 2) moves back to timeframe 2.

5.2.7.3 Bus-padding

Bus-padding is required in 3-state circuits to avoid that random padding, after the test sequence has been generated, will cause bus-conflicts (see Section 3.2.2, Page 45).

Normally, bus-padding is performed after a test for a fault has been generated. The motivation for this approach is that bus-padding should not influence the test generation process; i.e., constraining the search space during test generation. If this approach is applied in STPG, the decision flow during STPG and bus-padding will be as shown in Figure 5.20a. After the test generation, bus-padding is performed, which starts in the last timeframe of the sequence (timeframe 3; see 'start point' of bus-padding in the figure). If bus-padding in that timeframe has been finished, the process continues in the timeframe before it. Objectives on the PPIs of timeframe 3 are implied on the corresponding PPOs in timeframe 2, and justified, etcetera. The disadvantage of this method is that when bus-padding in a certain timeframe fails (the decision stack for bus-padding is exhausted), it is not easy to backtrack to those decisions which may have caused that bus-padding in that timeframe had failed. E.g., when bus-padding in timeframe 2 fails, then decisions made in phase 3 or phase 2 in timeframe 2, and/or earlier made decisions in other timeframes, have to be remade. To remake decisions of phase 3, all decisions of phase 3 in timeframes
5.2. Deterministic test generation

Figure 5.21: Bus can only be conflict-free padded with assignments to the PPIs. Hence an asynchronous reset state is needed

1 and 4 have to be destroyed; they depend on the decisions made in timeframe 2, and can not have any influence on bus-padding in timeframe 2. When a decision of phase 3 in timeframe 2 has been remade, then first state-justification has to be performed (phase 3); i.e., new decisions in timeframes 1, 4 and possibly more timeframes will result. After that, bus-padding can restart in timeframe 2. It is clear that this is not efficient.

Therefore, we propose to perform bus-padding during the STPG process. Figure 5.20b shows the decision flow for this approach. After phase 3 (backward state justification phase) for a timeframe has been finished, bus-padding is performed. The advantage of this method is that the “normal” decision flow is maintained; there is no difference in decision flow between Boolean and 3-state circuits. When bus-padding fails in a certain timeframe, backtracking can immediately be performed on decisions of phase 3 in this timeframe, as shown in the Figure.

Some 3-state sequential circuits require an asynchronous reset to allow for the generation of bus-conflict free sequences. Figure 5.21 shows a multiplexer. The data inputs and the control input of this multiplexer are PPIs. To assure that the bus in the multiplexer is conflict-free, at least one of the PPIs has to be assigned a value (0 or 1). Hence, it is not possible to have a timeframe without requirements on the PPIs; i.e., the initial state of this circuit cannot be the unknown state. This circuit needs an asynchronous reset-state; otherwise, STPG cannot be performed for this circuit.

To determine if a 3-state circuit requires a reset-state, the following procedure is performed, within one timeframe, before test generation:

Step 1: Perform bus-padding, assuming the PPIs uncontrollable. If this fails, than the buses cannot be conflict-free padded, and hence, an asynchronous reset-state is required. Otherwise, bus-padding is possible without requirements on the PPIs and no reset-state is necessary; the procedure is finished.

Step 2: (If step 1 has failed:) Perform bus-padding, within one timeframe and assuming the PPIs controllable. If this fails, than the circuit cannot be padded without bus-conflicts. Otherwise, the requirements on the PPIs are defined as the asynchronous reset-state of the circuit.

5.2.8 Backtrack procedure

This subsection proposes two new techniques, related to the backtrack procedure, to facilitate some form of (conflict-directed) back-jumping [Pro93] [Gas79]. Incorrect decisions
made during STPG, may result in a huge amount of backtracks, when between the conflict detection and the incorrectly made decisions, many other decision have been made. Figure 5.22 shows an example. Two incorrect decisions have been made on the PI and PPI in timeframe 2. This is detected in timeframe \( n + 1 \), and many backtracks may be required before those incorrect decisions are remade. It would enhance the performance of the STPG when decisions can be discarded without remaking, based on the idea that these decisions did not cause the conflict; i.e., discard decisions without losing completeness of search in STPG. The STPG jumps immediately to decisions which are likely to have caused the conflict in STPG, and remakes these decisions. In the next two paragraphs, we propose two techniques that provide some level of back-jumping.

**Backtrack until the PPI state has been changed.** When the STPG proves that a PPI state can not be justified (hence it is illegal and stored in the state restrictor), or the current PPI state is identified as being illegal by the state restrictor, the backtrack procedure starts on the current timeframe. Because the reason for the conflict is the PPI state, the backtrack procedure discards all decisions and restores the values on the signal lines to their value before these decisions were made, until the current PPI state changes. It does not check the three-bit flag, which denotes the already tried alternatives of these decisions. Figure 5.23 shows a decision stack of a timeframe. The PPI state of this timeframe is identified as being illegal. Assume that (in Figure 5.23), the PPI state changes, after discarding three decisions; decisions 5, 4, and 3 (and restoring the values on the signal lines to their values, before these decisions were performed). Hence, decision 2 is not discarded, but remade.

**Backtrack until all timeframes after the current timeframe have been removed.** When the STPG detects that a PPI state in a certain timeframe is illegal, backtracking
5.3. Test generation strategies

Figure 5.24: Decision stacks of timeframes $n - 1$, $n$, $n + 1$, \ldots, $n + k$. All timeframes after timeframe $n$ can be destroyed when the illegal PPI state $n$ is not changed after the decision stack of phase 3 in timeframe $n$ is exhausted.

Figure 5.25: a) a deep circuit; b) a wide circuit

has to be performed until the PPI state has been changed (see previous paragraph). Figure 5.24 shows the decision stacks of timeframes $n - 1$, $n$, $n + 1$, \ldots, $n + k$. The PPI state of timeframe $n$ has been identified as illegal, and therefore backtracking is performed (the STPG is currently in the backward justification phase, phase 3). Assume that all decisions made in phase 3 do not change the PPI state; i.e., the PPI state results from decisions made in phase 1 or 2 in timeframe $n$, and hence, these decisions have to be remade. In that case, all timeframes generated after timeframe $n$ (i.e., timeframes $n + 1$, $n + 2$, until the timeframe where the D-front had reached a PO (timeframe $n + k$)) can be destroyed; this is shown in the figure. After destroying these timeframes, the backtrack procedure can continue in timeframe $n$ with the decision stack of phase 1 or 2.

5.3 Test generation strategies

Just like in real life and in many games, applying different strategies, on different situations, can result in a higher performance. Also in ATPG, strategies are used to improve the performance. The selected strategies depend on properties of the circuit: a circuit can be very deep or very wide (see Figure 5.25), have many or few ISes, is sequential or combinational, has 3-state elements or not, etc.
In this section, strategies are discussed which can be applied in STPG. These strategies aim at improving the performance of the STPG. The strategies are selected based on properties of the circuits, or they are selected based on their history of successfulness so far during STPG. The “best” strategy (so-far) is always tried first on a new target fault. A strategy is selected as “best” when the STPG, using this strategy, has been successful for a certain number of faults in a sequence. This way it is possible that during STPG for one circuit, different strategies can become “best” at different times, depending on different groups of faults.

The following subsections present eight strategies.

### 5.3.1 Fault propagation before justification of objectives

When this strategy is enabled in phase 1 or 2, the STPG will first handle objectives related to fault propagation. After the fault-effect has been propagated to a PO/PPO, the objectives for justification are satisfied. Usually, the FAN-based STPG handles objectives for justification and fault propagation simultaneously (see Section 3.2.1, Page 41). However, for some circuits this can result in large lists of objectives. These large lists of objectives, which are handled by the multiple backtrace, can result in many conflicting requirements in the circuit (e.g., both values 0 and 1 are required on a fanout stem), or result in useless objectives (see Section 3.2.1, Page 43). The conflicting requirements or useless objectives would not, or less frequently occur when the number of objectives handled in parallel is reduced. This strategy separates the group of objectives for (fault) justification from those for fault propagation, and hence limits the number of objectives which are handled simultaneously by the backtrace procedure. In addition, the strategy gives an alternative search order, which is profitable by itself.

### 5.3.2 Order of search during backtrace

This strategy switches the search order on the inputs of elements. Figure 5.26 shows a NAND gate with value 1 requested on the output. Based on the controllability costs $MC^0$, inputs $a$ and $d$ are candidates, to be assigned value 0, to justify the output value of this element. Depending on the order of search over the inputs (from input $a$ to $e$, or from input $e$ to $a$), input $a$ or $d$ is selected.
5.3. Test generation strategies

5.3.3 Single backtrace

This switch enables the Single Backtrace (SB) procedure during STPG, instead of the Multiple Backtrace (MB). The SB starts with one objective and traces this objective back to the headlines, resulting in one decision on a headline. A single objective is selected from the list of unjustified lines\(^4\). In Section 3.2.1, Page 42 the Single Backtrace procedure (SB) has been explained in detail. Conflicting requirements on signal lines cannot occur during the SB. Our experience is that STPG for circuits which are highly testable (i.e., have a high fault coverage), performs on average better with SB than with MB (see also Page 42). In addition, especially circuits which are deep\(^5\) and/or have many Fanout-Free Regions (FFRs) (see Figure 5.27) with a large number of inputs, can have much advantage of this strategy. E.g., ISCAS circuits s641 and s713, which are deep circuits (> 70 levels), perform much better with SB than with MB (see experimental results in the next chapters). The FFRs of the ISCAS circuits s820 and s832 have relatively many inputs (FFRs with more than 40 inputs). These two circuits perform also better when SB is applied (in addition, these four ISCAS circuits are very testable).

5.3.4 Targeted D-front propagation

Targeted D-front propagation, proposed by [Nie91], has been described on Page 67. During STPG, in phase 1 and 2, the "best" D-front (based on observability cost estimates) is selected for fault propagation, the remaining D-front elements are stored on the decision stack (see Figure 5.28). A single D-front element instead of a (large) list of D-front elements may increase the number of mandatory assignments (because there may be more dominators), and hence, may increase the fault efficiency (i.e., less aborted STPG runs), because less (possibly incorrect) decisions are required. This strategy has been identified as very useful for circuits which are wide and/or have elements with large fanout. Such circuits often have very large D-fronts.

A disadvantage of this strategy is that the decision stack can become very large (as shown in Figure 5.28): From a D-front, the "best" one is chosen for fault propagation, and the remaining D-front elements are stored as alternative decisions on the decision stack. So, when the D-front is large, many D-front decisions are stored on the stack, which all

---

\(^4\)Selection is performed in circuit level order; first the highest leveled unjustified lines.

\(^5\)A circuit with more than 50 levels is defined as a "deep" circuit (experimentally determined).
may have to be selected as alternative “best” D-front element in turn, during backtracking. This increases the number of backtracks for many faults; proving a fault to be untestable may become very expensive. Therefore, we can apply an alternative method of targeted D-front propagation, as a strategy: Instead of selecting the “best” D-front element each time the number of D-front elements is larger than one, the selection procedure is only performed when STPG starts a new timeframe and the D-front is initialized. I.e., the number of times targeted D-front propagation is applied is constrained. This saves a lot of D-front decisions, reducing the decision stack size. As a result the fault efficiency of a circuit may increase because more untestable faults are identified.

5.3.5 Find PPI state in previous generated test sequence

This strategy has been proposed in HITEC [Nie91] (see Section 4.4, Page 68). Consider the STPG process being in phase 3 (backward state justification), having completed a timeframe, and going to select the next timeframe to justify the current PPI state. However, when this PPI state covers one of the states in the test set, it may save computing time, when the input sequence stored in the test set, which results to this covered state, is added before the current generated sequence.

However, the fault is not guaranteed detected when this strategy is applied: The fault(-effect) has not been considered in the timeframes of the added input vectors, and this may invalidate the test sequence. Therefore, after STPG, fault simulation is performed to check if the fault is detected. In the case the fault is not detected, the fault is marked aborted and has to be selected as fault target again in the next ATPG stage.

5.3.6 Ignore PPO reached, when PO is reachable in timeframe

This strategy aims at reducing the length of the generated test sequence. Usually, the fault propagation within a timeframe is completed when a PPO has been reached and the PPO state is not equal to any previously generated PPO state (see Page 86). However,
5.3. Test generation strategies

![Diagram](image.png)

**Figure 5.29:** a) After STPG has finished phase 1, it preliminary tries to justify PPIstate1A to check if initialization of PPIstate1A is possible; b) After the justification of PPIstateA has been succeeded, fault propagation is performed. If PPOstate1 is covered by PPIstateB the initialization sequence of PPIstateA is re-used, otherwise a new one is generated.

sometimes still POs can be reached in the current timeframe. When this situation occurs (and this strategy is enabled), fault propagation in the timeframe is continued, until a PO has been reached. The advantage of this strategy is a reduced test sequence length. However, experimental results have shown, that for some circuits the fault efficiency decreases. Therefore, this strategy is not used by default, but only as an alternative search strategy.

5.3.7 State justification before fault propagation

The STPG, applying this strategy, performs a preliminary state justification phase (phase 3) after phase 1 has been finished, and before phase 2 is started (assuming phase 2 is required; i.e., a PO has not been reached in phase 1). Figure 5.29a shows an example: The STPG tries to (preliminary) generate an initialization sequence for PPIstate1A. After the STPG has been successful, the generated initialization sequence is stored, and STPG continues the normal generation procedure (i.e., start with phase 2).

When STPG has completed the fault propagation phase, the resulting PPIstate1B (see Figure 5.29b) is compared to the reached PPO state of the preliminary generated initialization sequence (PPOstate1; Figure 5.29a). If PPIstate1B covers PPOstate1, STPG has been finished. If it does not cover PPOstate1, the preliminary generated initialization sequence is destroyed, and a new one is generated by STPG. Figure 5.29b shows that PPIstate1B does not cover PPOstate1, hence a new initialization sequence has to be generated. If PPOstate1 covers PPIstate1B, then PPIstateB has additional requirements which have to be justified. Also in that case, the preliminary initialization sequence is destroyed, and a new one is generated, because we cannot backtrack on the decisions made for the preliminary initialization sequence. Note, that it is not allowed
to simulate value 1, part of PPOstateI (see Figure 5.29a) in timeframe 1 (and further),
because this will (incorrectly) constrain the search space for phase 2.

The advantage of this strategy is that many untestable faults are identified; the STPG
proves that PPIstate1A cannot be justified (resulting in a GIS for the State Restrictor).
Using the normal STPG procedure (where this strategy is disabled), the STPG can fail
to handle these faults because of time constrains. Phase 2 (and/or the justification of
PPIstate1B) consumes too much time/many backtracks, resulting in an aborted fault.
Another advantage of this strategy is that during the justification phase of PPIstate1A,
GISes may be identified. Because PPIstate1B is always covered by PPIstate1A, these
GISes may be more useful for the justification process of PPIstate1B, than other GISes,
identified in previous STPG runs.

5.3.8 Recursive learning

Finding as many as possible mandatory assignments during the STPG process is necessary
to reduce the complexity of STPG, because mandatory assignments will reduce the size
of the search space. The strategy described in this section, called recursive learning, aims
at finding new mandatory assignments, which would not be identified by local reasoning.
It is a very powerful technique, however, it is an expensive (in computing time) strategy.
Therefore, the STPG only performs recursive learning when other strategies have failed.

Recursive learning has been originally proposed as a stand-alone technique for gener-
ating test patterns [Kun92]. In [vdL96], the use of recursive learning has been proposed as
a powerful additional tool for finding mandatory assignments during TPG for a fault. We
use recursive learning in a similar way. However, instead of performing recursive learning
after each new decision during the test generation process [vdL96], recursive learning is
performed when a new set of initial objectives has to be created for the backtrace (i.e., flag
make_new_objectives is set; see Page 86). This limits the computing time consumed by
this technique (the total number of decisions is usually much higher in STPG compared
to CTPG; in STPG, in every timeframe decisions can be made).

Two recursive learning procedures are applied during test generation: justification and
propagation learning.

Justification learning is performed to find mandatory assignments for unjustified values,
uv’s (i.e., output values of elements which are not (yet) supported by these elements’
input values). For each uv, all possible assignments (justification options) are performed
one by one (temporarily) to justify the uv. For instance, to justify the uv 0 at the AND
gate output in Figure 5.30a, each input is temporarily assigned a 0. Each temporary
assignment, results in an implication set (set of implied values). The intersection of these
sets yields mandatory assignments for the uv; in the figure, k=0 is learned. Temporary
assignments made to justify a uv may result in new uv’s, allowing a recursive call to
the justification learning procedure in order to extend the implication sets of justification
options, as shown in Figure 5.30b, where k=1 is learned.

In Section 5.4, we propose to use justification learning during the static learning pro-
cess, in order to increase the number of identified global assignments.
5.4 Static learning to support STPG

Figure 5.30: [vdL96]a) Justification learning: $k=0$ is in the intersection of implication sets for all justification options; b) 2 levels deep justification learning

![Diagram showing justification learning](image)

Figure 5.31: [vdL96]Propagation learning

Propagation learning is performed to find mandatory assignments for propagation of the fault-effect to the (P)POs. For each D-front element, one at a time, all assignments are performed to propagate (the fault-effect) through this element. These assignments and their implications are stored in sets. The intersection of these sets yields the mandatory assignments for propagation of the fault-effect. For example, in Figure 5.31, for both D-front elements the set of implied values (for propagation through such a D-front element) is determined, and the intersection of these sets contains a 0 on the fanout-stem (line a), which is thus a mandatory assignment. Temporary assignments made to propagate current D-front elements may result in v’s causing a call to the justification learning procedure, further extending the implication sets of the propagation options.

5.4 Static learning to support STPG

In this section, learning techniques are introduced, which are performed during the ATPG process to lighten the task of, and accelerate, the test generation process. Three static learning techniques exist, which can be performed before STPG is started:

1. Static learning based on logic implication. In Subsection 5.4.1, prior art on this learning technique is described, and a new learning technique is proposed.

2. Static learning based on signal line justification. This learning technique identifies
signal lines which are uninitialized to 0 or 1. In Subsection 5.4.2, a learning procedure is proposed based on this technique.

3. Static learning based on state identification. This technique is applied to identify GISes. In Chapter 7 GIS identification is presented; therefore, this form of static learning is presented in that chapter.

5.4.1 Static learning based on logic implication

Static learning, based on logic (exhaustive) implication, aims at learning \textit{global} or \textit{indirect} implications; i.e., those (mandatory) assignments that cannot be found by simple forward and backward implication (\textit{local} or \textit{direct} implications). Schulz \textit{et al}. [Sch87] [Sch89b] proposed learning using two binary logical inference rules: The rule of contraposition, and the rule of constructive dilemma (these rules are explained later in this subsection). For each signal line, once a 0, and once a 1 are exhaustively implied in circuit; after each implication, both rules are applied on the assigned values, to learn global implications. In [vdL96] static learning has been extended for 3-state combinational circuits, and in [Sch89a], it has been extended for sequential circuits. This last extension enabled the identification of global implications over timeframe boundaries during STPG. Other methods to perform learning based on logic implication have been proposed by Kunz \textit{et al}. [Kun92] (recursive learning, see Section 5.3.8) and Zhao \textit{et al}. [Zha97]. Zhao proposed a learning technique based on set-algebra and the transitivity law. Similar to deductive fault simulation [Ams72], where sets of faults are propagated from inputs of a gate to its output using set operations of intersection, union, and difference, this method propagates sets of implications from inputs of a gate to its outputs. Furthermore, they integrated into the set operations, the rule of transition ($a \rightarrow b \land b \rightarrow c \Rightarrow a \rightarrow c$) and contraposition. Using this method they find much more global implications than the static learning, described in [Sch89b]. However, similar to deductive fault simulation, the method can be computing time expensive due to the set operations over all signal lines. Hence, this method is not suitable for large, industrial circuits.

In this subsection, a new learning technique is proposed, based on the static learning procedures in [Sch89b], [Sch89a], and [vdL96], into which we have integrated the justification learning procedure of recursive learning [Kun92]. This new static learning procedure identifies much more global implications than have been identified in [Sch89a], [vdL96]. In addition, global implications over timeframe boundaries are identified. An advantage of our method is that it can also be applied to large (sequential) (3-state) circuits.

The remainder of this subsection is organized as follows: In the first paragraph, static learning for combinational circuits is presented. Thereafter, in the second paragraph, the modifications for static learning in sequential circuits are described. In the third paragraph, the properties of the global implications performed during STPG are described. Finally, in the last paragraph, experimental results are presented, which are compared to the results from [Sch89b] and [Zha97].
Static learning for combinational circuits. For each signal line $S$, once a 0 (denoted as $(S = 0)$) and once a 1 $(S = 1)$ is backward and forward implied, resulting in two sets of assignments: $S_0$ and $S_1$. If there are unjustified values after $(S = 0)$ or $(S = 1)$ have been implied, justification learning is performed on these unjustified values (recursive learning, see Section 5.3.8). The mandatory assignments learned from justification learning are added to respectively $S_0$ and $S_1$. If during exhaustive implication of $S = sv$; $sv \in \{0, 1\}$ conflicts have been detected then the implication is impossible; i.e., $S$ is identified as a signal line fixed to value $sv$. Figure 5.32 shows a part of the ISCAS circuit c6288; this circuit is used to demonstrate the techniques applied in static learning. E.g., assignment of value 1 on line $h$ results in a conflict; i.e., line $h$ is fixed to 0.

The following terms are used to represent signal lines and assignments: $c_o$ denotes a signal line in the set $S_o$. $sv(c_o)$ denotes the assigned value on signal line $c_o$. $(c = sv)$ denotes the assignment of $sv$ on line $c$. The rules of contraposition and constructive dilemma are applied on the lists $S_0$ and $S_1$. Only useful (i.e., global) implications are learned and stored. The rules are applied in the following way:

- **constructive dilemma** $((a \lor b) \land (a \rightarrow c) \land (b \rightarrow c) \rightarrow c)$: Figure 5.33b shows how this rule is applied: $(S = 0) \lor (S = 1) \land ((S = 0) \rightarrow (c = 1)) \land ((S = 1) \rightarrow (c = 1)) \rightarrow (c = 1)$; i.e., irrelevant of other assignments in the circuit, signal line $c$ has value 1. This line with corresponding value is stored in a list of fixed lines.

- **contraposition** $(a \rightarrow b \iff \bar{b} \rightarrow \bar{a})$: This rule is applied between the signal line $S$ and each assignment stored in the two sets $S_0$ and $S_1$. Figure 5.33a shows how this rule is applied between line $S$ and $c$: $(S = 0) \rightarrow (c = 0) \Rightarrow (c = 1) \rightarrow (S = 1)$; i.e., the assignment $(S = 1)$ is a global implication when line $c$ gets value
1. However, not each combination of line $S$ and a line in the set results in useful learning. E.g., it is not useful to learn: $(S = 0) \rightarrow (a = 1) \Rightarrow (a = 0) \rightarrow (S = 1)$ because this implication is a local implication. Therefore, three rules are defined, to determine if a learned implication is useful to be stored as a global implication. A learned implication is useful when $(L(c)$ denotes the level of the line in the circuit):

1. $L(S) \leq L(c)$ and the line $S$ is not one of the inputs of the element which drives line $c$ [Sch89b]. Based on this rule, the implication $(c = 1) \rightarrow (S = 1)$ is useful to be learned and $(a = 0) \rightarrow (S = 1)$ is not (see Figure 5.33a).

2. $L(S) > L(c)$ and there is no path from line $c$ to line $S$ [Sch89b].

3. $L(S) > L(c)$ and value $sv(c)$ on line $c$ can not propagate to line $S$ via the dominance property of Boolean elements. E.g., assignment $(c = 0)$, in Figure 5.32, can propagate to line $j$ via the dominance property; so the implication $(c = 0) \rightarrow (j = 0)$ is not a global implication, and therefore not useful to be learned. Another example: $(k = 1) \rightarrow (c = 1) \Rightarrow (c = 0) \rightarrow (k = 0)$; although there is a path from $c$ to $k$, $(c = 0)$ cannot propagate to $k$ via dominance; i.e., $(c = 0) \rightarrow (k = 0)$ is useful to be learned.

Using the rules of constructive dilemma and contraposition, and recursive learning, static learning has become very powerful; many indirect implications are identified, which are not identified in the original static learning procedure, proposed by [Sch87]. For example, in the circuit shown in Figure 5.32 the following implications are learned:

1. $(k = 1) \rightarrow (c = 1) \Rightarrow (c = 0) \rightarrow (k = 0)$
2. $(a = 1) \rightarrow (c = 1) \Rightarrow (c = 0) \rightarrow (a = 0)$
3. $(s = 1) \rightarrow (c = 1) \Rightarrow (c = 0) \rightarrow (s = 0)$
4. $(a = 1) \rightarrow (a = 1) \Rightarrow (a = 0) \rightarrow (a = 0)$

Learnings 1, 2 and 3 are also reported in [Zha97]; however they do not detect the fourth one as an indirect implication. Furthermore, in [Zha97] two iterations over all signal lines of the circuit were required to learn the first three implications. Our method required only a single iteration.

After static learning has been performed, an iteration is performed over the signal lines of the circuit to remove redundant learnings, to speed-up the implication procedure during STPG. Using the transitive law $(a \rightarrow b, b \rightarrow c \Rightarrow a \rightarrow c)$, many learnings can be identified as redundant: When $a \rightarrow c$, and $b \rightarrow c$ are learned (i.e., global implications), and $a \rightarrow b$ is also an implication, then the learning $a \rightarrow c$ is redundant and can be removed. In Figure 5.32, the following two implications are learned: $(r = 1) \rightarrow (n = 0)$ and $(q = 0) \rightarrow (n = 0)$. However, $(r = 1) \rightarrow (q = 0)$ is a local implication; hence, the learning $(r = 1) \rightarrow (n = 0)$ is redundant and can be removed.

**Static learning for sequential circuits.** The static learning method, described in the previous paragraph, is also performed over timeframe boundaries in sequential circuits. In a limited number of timeframes, static learning is performed: For every signal line $S$
in timeframe 0, \((S = 0)\) as well as \((S = 1)\) is exhaustively implied. When the implication procedure reaches PPIs (PPOs), the values on these PPIs (PPOs) are implied in the previous (next) timeframe on the corresponding PPOs (PPIs). We have limited the total number of timeframes to five. Figure 5.34 shows these five timeframes: Implications, two timeframes forward and two timeframes backward relative to timeframe 0, are allowed. Implications in timeframes more than two before (after) timeframe 0, are usually not useful (if there are any): They often repeat the implications also performed in timeframes \(-1\) and \(-2\) (1 and 2), due to the (feedback-)loops in the circuit.

The rules of constructive dilemma and contraposition are applied to the resulting sets \(S_0\) and \(S_1\). However, the assignments in these sets may have been performed in different timeframes then the timeframe where the value of \(S\) has been assigned. \(RT(c)\) is defined as the relative timeframe number of the assignment on line \(c\), compared to the timeframe, where \((S = sv)\) has been performed. To determine if a learned implication using contraposition is useful to be stored (i.e., it is a global implication), the following three rules are defined, depending on \(RT(c)\):

1. \(RT(c) = 0\): Apply the three rules, as described for static learning for combinational circuits, to determine if the implication is useful to be learned.

2. \(RT(c) > 0\): The learned implication is always useful.

3. \(RT(c) < 0\): Determine all PPIs of timeframe 0, which are reachable from line \(c\); i.e., there is a path (possibly over timeframe boundaries) from line \(c\) to these PPIs.

Then for each reached PPI apply rule 1. The learned implication is not useful as soon as one PPI does not satisfy rule 1.

A large number of global implications is identified using this static learning method.

**Performing global implications during STPG.** The assignment of a value on signal lines may result in global implications in the circuit. The signal line, which causes global implications, is denoted as \(S\).

**Backward Global Implications** (BGIs) (see \(g1\) and \(g3\) in Figure 5.35) are implications on signal lines, which have a circuit level lower than the level of \(S\) (\(g1\)), or are performed in one of the previous timeframes (\(g3\)). BGIs have to be justified, to justify the cause of the BGIs, signal line \(S\). In other words, the BGIs are required for signal line \(S\); therefore, BGIs have to be justified.

**Forward Global Implications** (FGIs) are implications on signal lines, which have a circuit level equal or larger than the level of \(S\) (\(g2\)), or are performed in one of the next
timeframes (g4). FGIs are assigned due to signal line $S$, but they are not required for justification of signal line $S$. Because signal line $S$ is assigned, the lines of the FGIs will become the value of the FGI, whether the FGI is performed or not. For example, the learned FGI, $(c = 0) \rightarrow (k = 0)$ in Figure 5.32 is not necessary to be performed; after STPG, when all signal lines have a specified value, line $k$ will have value $0$ (due to $(c = 0)$), anyway. The only reason that FGIs are performed is to increase the number of mandatory assignments during STPG. Hence, FGIs (in the current timeframe (g2) and next timeframes (g4)) do not have to be justified; i.e., are not objectives during STPG. Because the FGIs are not explicitly justified by the STPG, a necessary requirement of the generated sequence is that all signal lines will have a specified value to implicitly justify the FGIs; i.e., all PIs of each timeframe must have a value after ATPG (e.g., by performing random padding after STPG).

**Experimental results.** The static learning algorithm, with recursive learning, has been implemented and has been incorporated into the DAT system. Static learning has been performed on a workstation with a Pentium Pro 200Mhz processor with 256 MB RAM. The maximum recursion depth during learning was limited to 1. This paragraph presents the experimental results for the ISCAS’85 combinational benchmark circuits, and compares these results with static learning based on the method proposed by [Sch87], and the
published results in [Zha97]. Table 5.4 shows the results of our static learning algorithm (DAT), and [Zha97]. For each circuit, the total number of global implications (column global), the total number of fixed lines (column fix), and the CPU time (column CPU) are shown. In addition, the number of useless learnings (column useless) is reported. For all experiments, the global implications on fanout elements have been counted as a single implication; i.e., we do not discriminate between stems and branches. In our experiments, buffer/inverter elements (chains) have been counted as a single implication; this has not been reported in [Zha97].

The new static learning procedure of DAT identifies much more global implications, compared to the results of [Sch87]. However, often fewer implications have been identified compared to [Zha97]. Note that often a large number of the global implications is useless/redundant; however, not reported in [Zha97], this also accounts for their results ([Zha97] uses the transitively law to find new global implications; however, especially those global implications often are useless). DAT identifies more fixed signal lines, and is (relatively) faster than [Zha97] (ratio between total number of learned implications is 2.3 and ratio between CPU times is 3.0). The CPU time differences between DAT and [Zha97] are larger for the larger ISCAS circuits. This is probably due to the larger exponential time complexity of [Zha97], compared to static learning with recursive learning.

5.4.2 Static learning based on signal line justification

This section presents a new method to identify signal lines which are uninitializable to 0 and/or 1. Many uninitializable signal lines have already been identified by applying the pre-process FUN, as described in Section 5.1.3, Page 81. However, some of them are not identified. Figure 5.36 shows an example. The output of the NAND is uninitializable to 0, because the inputs cannot have value 1 at the same time. FUN detects that each input can become value 1, and concludes (incorrectly) that the output can become value 0. This learning method aims at finding all of these uninitializable signal lines. The procedure is as follows:

Procedure: Signal line justification.
For each signal line $S$ in the circuit perform:

1. if not has been $0(S)$ and not uninitializable to $0(S)$
   and not fixed to $1(S)$ then

   (a) Justify signal line( $S$, 0).
2. if not has\_been\_1( S ) and not uninitializable\_to\_1( S )
   and not fixed\_to\_0( S ) then
   
   (a) Justify\_signal\_line( S, 1 ).

   (b) if result is unjustifiable then Mark\_Uninitializable( S, 1 )
       else if result is justifiable then mark\_has\_been\_1( S )

All signal lines, which have been value 0 (1) prior to execution of this procedure (e.g.,
they have been assigned 0 (1) during STPG or fault/logic simulation, which can have
been performed before this procedure), or which have been marked uninitializable to
0 (1) (due to pre-process FUN), or have been fixed to 1 (0) (due to static learning to
find global implications), are skipped. The remaining signal lines are tried to be justified,
using the STPG’s justification procedure (phase 3). Signal lines which cannot be justified
are marked as uninitializable to 0 (1).

5.5 Fault simulation

In this section, the Fault Simulation (FS) algorithm for sequential circuits as it is used in
DAT is presented. The FS can be classified as a single pattern, single timeframe, single
fault, explicit fault simulator. This simulator has been developed to validate the results
of the STPG, and to find additionally detected faults in a vector sequence, that has been
generated by the STPG.

The FS is based on repeated logic simulation per timeframe: Once, an input vector is
simulated to establish all the 'good' signal values; thereafter for each signal line $l$ (whose
Corresponding fault is not yet detected), a fault is injected by assigning the 'faulty' signal
value, and this value is propagated through the circuit by means of simulation. The FS
algorithm is described hereafter and illustrated in Figure 5.37.

Algorithm: Fault simulation

For each sequence (test pattern) in a the test set:

- For each vector in the sequence:
5.6. Additional target test generation

1. Assign vector on the PIs of a timeframe.
2. If (first vector of sequence): Initialize PPIs to value $U$; else: Copy PP0state(previous timeframe) to PPIs.
3. Perform logic simulation.
4. For each still undetected fault $F$ in the fault list perform:
   (a) Inject the fault $F$ in the circuit.
   (b) Inject the fault-effects of fault $F$ on the PPOs of the previous timeframe (if any), on the corresponding PPIs.
   (c) Perform event-driven logic simulation.
   (d) As soon as a PO has become fault sensitive, the fault $F$ is marked detected.
   (e) If fault $F$ is not detected: Store this fault's fault-effects on PPOs (if any) in a list, to be used for the next timeframe.
   (f) Restore fault sensitive signal lines to their fault-free value.

Figure 5.37 shows two timeframes with fault $F$ and its fault sensitive paths. Assume that there is one test pattern, a sequence consisting of 2 test vectors, and the fault list consists of one fault ($F$). The initial state is, in this case unknown, but can be any reset-state. Test vector 1 is assigned on the PIs of the first timeframe (step 1, in the Algorithm), the unknown initial state is assigned to the PPIs (step 2), and logic simulation is performed (step 3). Fault $F$ is injected (step 4a) and (event-driven) simulated (step 4c). Assume that fault $F$ is not detected on a PO (step 4d). Three PPOs have fault-effects from fault $F$ (i.e., they are fault sensitive). These PPOs and their faulty values are stored in a list for fault $F$ (step 4e). Note that, storage of the faulty value of the PPO is necessary, because the faulty value of the PPO is not necessarily the inversion of the fault-free value (1/0 or 0/1), but can also be $U$/0, $Z$/1, etc. Thereafter, all signal lines which has been changed due to the fault $F$ are restored to their original value (step 4f). Usually, now the next fault is selected (step 4). However, in this single fault example, we continue by selecting the next vector of this sequence. Test vector 2 is assigned on the PIs of the second timeframe (step 1), the PPO state of timeframe 1 is copied to the PPIs of timeframe 2 (step 2), and logic simulation is performed (step 3). Then fault $F$ is injected (step 4a), the fault-effects on the PPOs of the previous timeframe (timeframe 1) for this fault $F$ are injected on the corresponding PPIs (step 4b). Then logic simulation is performed (step 4c). A PO is detected as fault sensitive, hence the fault is detected (step 4d). The sensitive PPO(s) in timeframe 2 are ignored, and the (faulty) value of these PPOs are not stored. Finally, the faulty values are removed in the current timeframe (step 4f). Thereafter, the next fault (if any) can be selected.

5.6 Additional target test generation

This section presents additional target test generation. Additional target TPG [Pom91a] [vdL96] usually aims at generating small test sets. When a test for a fault has been found,
an additional target is selected in order to extend the earlier test to cover the additional target. The target of additional target TPG, within our context, is to increase the fault coverage. Of course, additional target TPG will also generate smaller test sequences, but that is (within our context) regarded as a pleasant side-effect, as long as high fault efficiencies for current circuits cannot yet be established.

The fault coverage of a test sequence, which still has unassigned entries, can often easily be increased by using additional target TPG. Because the additional targets may make use of the (fault-)sensitive paths in this sequence, which have been generated for the primary, and previous successfully added additional targets. Often, a few extra decisions on the headlines are sufficient to extend the test sequence to detect additional targets, (possibly) along these sensitive paths.

Within the context of STPG, we define additional target TPG as a method to extend an existing test sequence (i.e., length of the test sequence may increase), for an additional target fault, in such a way that it will test that fault. Based on this definition, three methods have been developed:

A1. Extend the current test sequence during ATPG, by selecting additional target faults from the set of faults which have fault-effects on the last generated PPO state (determined by FS), after a sequence has been made (or extended) for an earlier fault.

A2. Extend the current test sequence after ATPG, by selecting additional target faults, from the set of aborted faults which have fault-effects on one or more previously generated states.

A3. Extend the current test sequence, after STPG for a primary target, by selecting additional targets from the set of undetected faults, as long as there are unassigned PIs, and there are targets.

Method A1 and method A3 are described in the next two paragraphs. Method A2 has already been discussed on Page 68 and therefore does not need further explanation at this point. In the third and last paragraph, experimental results are given, to show the usefulness of STPG with method A1 or method A3. Method A2 is not applied during our ATPG processes.

Method A1: Extend the current test sequence by making use of fault-effects on the last PPO state.

A generated test sequence for the target-fault is usually simulated by the FS (see Section 5.5). The target-fault, and often a number of other faults may have been detected by this test sequence. Also, a number of faults has fault-effects on the PPOs in the last timeframe (see Figure 5.38). This method selects one of these faults as an additional target for the next STPG process. This process aims at propagating the fault-effect on the PPOs of the last generated timeframe to the POs, by generating additional timeframes. For example, Figure 5.38 shows that two additional timeframes have been generated to detect the fault F on the POs.

The main advantage of this method is that phase 1 of STPG (fault activation phase) and partially phase 3 (backward state justification phase) are skipped for a fault. The fault
5.6. Additional target test generation

Figure 5.38: Extend the test sequence by making use of fault-effects on the last generated PPO state

Figure 5.39: Additional target STPG after a test sequence for a primary target has been generated

is already activated, and usually the PPO state of the last generated timeframe is fully specified (i.e., the FFs have values 0, 1 or U; see Figure 5.38), and is justified. Hence, only phase 2 (fault propagation phase) is performed to propagate the fault-effect, from (a) PPO(s) to the POs, and phase 3 is performed to justify the PPI states of the new additional timeframes.

Method A3: Extend the current test sequence by making use of unassigned PIs in the current test sequence.

Figure 5.39 shows a test sequence consisting of three vectors, which has been generated for a primary target. Often, one or more PIs are still unassigned; assignment of these PIs was not required to detect the target-fault. Usually these unassigned PIs are randomly assigned a value (random padding) to increase the fault coverage of this sequence.

This method tries to assign the unassigned PIs deterministically; i.e., by extending the current sequence, to cover additional targets, using the STPG, usually resulting in a larger increase of the fault coverage of this sequence than when random padding is applied.

The method selects an undetected fault, for which the corresponding signal line in the leftmost timeframe (timeframe 1 in Figure 5.39) is not yet assigned, or whose value already activates the fault. The PPI state of this timeframe frequently is not fully assigned, hence this timeframe frequently has a large fraction of unassigned signal lines, increasing the chance that faults can be “added”. Phase 1, 2, and 3 of the STPG are performed as
usual in the current sequence (timeframe 1, 2 and 3), but given the assignments of current sequence as “fixed values” that cannot be redecided, to activate the fault and propagate the fault to a PO. This might increase the sequence length; i.e., new timeframes may be generated before timeframe 1 or after timeframe 3. If all PIs are assigned in the current timeframe (timeframe 1), or there are no faults anymore, the method moves to the next timeframe with unassigned PIs; i.e., moves to timeframe 2 in Figure 5.39. This process continues until the last timeframe of the sequence has been reached, or all faults have been detected.

**Experimental results.** In this paragraph, experimental results are presented to show the usefulness of additional target STPG in increasing the fault efficiency of a number of ISCAS’89 circuits. Two experiments have been performed on a PC with Pentium Pro 200 Mhz processor and 256Mb main-memory, under the Linux operating system.

The first experiment performed STPG without, and with additional targets, based on method A1. Table 5.5 shows the results. The results show that method A1 results in an increase of the fault efficiency for almost all circuits. The CPU time differences between both experiments are marginal. The number of test vectors of STPG with method A1 is increased, when the fault efficiency is increased. However, a test length reduction was expected, because the generation of initialization sequences is skipped (part of phase 3, the backward justification phase). A possible explanation of this behavior is the restriction imposed by the fully-assigned PPO state of the last generated timeframe of a test sequence for a fault. This may result in additional generated test sequences for (additional target) faults, which are on average longer than when the test sequences were generated for these faults as primary targets. A fault efficiency reduction due to method A1 is hardly noticed, except for circuit s1423. Based on the often improved fault efficiencies and/or CPU consumption, we can conclude that method A1 is an useful strategy for the ATPG process.

In the second experiment, STPG has been performed without, and with additional targets, based on method A3. Table 5.6 shows the results. These results show that STPG
Table 5.6: STPG without/with additional targets (method A3)

<table>
<thead>
<tr>
<th>Circuit</th>
<th>#levels</th>
<th>#vecs</th>
<th>feff</th>
<th>CPU</th>
<th>#vecs</th>
<th>feff</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>s298</td>
<td>9</td>
<td>130</td>
<td>99.68</td>
<td>276.98</td>
<td>199</td>
<td>99.68</td>
<td>346.18</td>
</tr>
<tr>
<td>s386</td>
<td>11</td>
<td>137</td>
<td>94.53</td>
<td>107.94</td>
<td>175</td>
<td>96.09</td>
<td>130.75</td>
</tr>
<tr>
<td>s526</td>
<td>9</td>
<td>708</td>
<td>89.91</td>
<td>742.62</td>
<td>355</td>
<td>80.54</td>
<td>116.85</td>
</tr>
<tr>
<td>s641</td>
<td>74</td>
<td>190</td>
<td>99.14</td>
<td>130.33</td>
<td>91</td>
<td>99.57</td>
<td>1322.3</td>
</tr>
<tr>
<td>s713</td>
<td>74</td>
<td>146</td>
<td>98.80</td>
<td>1481.5</td>
<td>87</td>
<td>99.48</td>
<td>1505.3</td>
</tr>
<tr>
<td>s820</td>
<td>10</td>
<td>436</td>
<td>82.00</td>
<td>800.44</td>
<td>439</td>
<td>83.76</td>
<td>1063.1</td>
</tr>
<tr>
<td>s1423</td>
<td>59</td>
<td>60</td>
<td>29.70</td>
<td>24925</td>
<td>45</td>
<td>40.6</td>
<td>21384</td>
</tr>
</tbody>
</table>

#lev: Number of circuit levels  
#vecs: Number of vectors in test sequence  
feff: Fault efficiency  
CPU: Computing costs

with additional targets perform well for circuits with a high number of circuit levels (circuits s641, s713 and s1423): The fault efficiencies have been increased, and the test set sizes have been reduced. STPG without additional targets, for these deep circuits have a low performance, because for those circuits, it is often a hard task for STPG to generate a propagating path from the fault to the POs; the fault-effect has to be traversed along many circuit levels. If STPG with additional targets did succeed in generating a propagating path for a primary target, re-use of this path is profitable, as the results show. Also for other circuits method A3 often performed well; the fault efficiencies of s386 and s820 have been increased; however, the test set size and/or CPU costs for these circuits increased. Sometimes the fault efficiency decreased when additional target STPG was applied; e.g., circuit s526; this can occur when during STPG without and with additional targets, a different set of GISes is identified; it is possible that during STPG with additional targets less GISes are identified because often illegal PPI states of a timeframe cannot be identified as a GIS: The timeframes have previous timeframes with assignments of previously detected faults.

5.7 Overview of the ATPG process

In this section, the ATPG process for sequential circuits is described based on the methods, techniques, and strategies, which have been described in the previous sections of this chapter. After the process has been presented, some comments are given on the choices made in the process related to the STPG strategies.

DAT ATPG process for sequential circuits

1. Read circuit design, and perform pre-processes (Section 5.1). Controllability and observability cost estimates for the signal lines are calculated. Untestable faults are identified based on uninitializable and unobservable lines, and combinational redundancy.

2. Static learning to learn global implications (Section 5.4.1).
3. Static learning to learn GISes (Chapter 7).

4. If circuit is deep (≥ 50 levels) or has large FFRs (i.e., average number of FFR inputs ≥ 6):
   Enable single backtrace (Section 5.3.3).
   Enable additional target STPG (method A3; Section 5.6). The backtrack limit during additional target STPG has been kept very low (4 backtracks).

5. Perform an ATPG stage using one STPG method (Section 5.2), with very low backtrack limits (usually between 256-512 backtracks).
   The strategies "fault propagation before justification of objectives" (Section 5.3.1), targeted D-front propagation (Section 5.3.4), and additional target STPG (method A1; Section 5.6) are enabled.
   Between the STPG runs, FS (Section 5.5) is performed to remove detected faults from the fault-list, and for the selection of an additional target to extend the sequence (method A1).

6. Static learning to learn unjustifiable signal lines (Section 5.4.2).

7. Perform three ATPG stages with increasing backtrack/time limits. Each ATPG stage has two STPG methods with different strategies enabled. Additional target STPG (method A1) is disabled, to have an alternative search method in STPG.
   Strategy "state justification before fault propagation" (Section 5.3.7) is enabled in one STPG method of the first and last ATPG stage. Often many untestable faults are identified by STPGs methods which apply this strategy.
   Strategy "order of search during backtrace" (Section 5.3.2) is enabled in a number of STPG methods over the ATPG stages, to influence the search during the backtrace.
   Strategy "ignore PPO reached, when PO is reachable in timeframe" (Section 5.3.6) is enabled in a number of STPG methods over the ATPG stages, to influence the fault propagation phase.
   Strategy "recursive learning" (Section 5.3.8) is performed in the second ATPG stage, to handle hard faults. To limit the computing time consumed by this strategy, the backtrack limit is kept low (128 backtracks).
   The STPG method which is most successful is tried first on a fault; i.e., an STPG method which has been successful two times in a sequence becomes the primary STPG method.

A circuit with more than 50 levels has been defined as a deep circuit. A circuit have "large" FFRs when the average number of inputs of these FFRs is larger than 6. Both values have been experimentally chosen, and are mainly based on the properties of the ISCAS'89 circuits.

In the previous sections we have described the advantages and/or disadvantages of the strategies. Strategies 1 (fault propagation before justification of objectives) and 4
(targeted D-front propagation) are enabled in most STPG methods. Although also these strategies do have disadvantages (see the specific sections), for most circuits they increase the performance of the STPG, or the performance reduction is not significant.

Strategy "find PPI state in previous generated test sequence" (Section 5.3.5) is usually disabled. This strategy may result in a reduction of CPU costs because less initialization sequences are generated. However, experiments have shows that this reduction is small.

Static learning to learn unjustifiable signal lines is performed after the first ATPG stage, because in that case, many signal lines already has been 0 and/or has been 1. Hence, the number of lines, which have to be initialized to 0/1 by this learning process, is reduced, saving computing time.

The experiments, described in the next chapters, have been performed on an ATPG system, which is based on this process.

5.8 Summary and conclusions

A state-of-the-art ATPG system for sequential 3-state circuit has been presented. Pre-processes have been described: Can_be_Z and can_be_U determination of the signal lines, to reduce the size of the search space. The combinational and sequential controllability cost estimates have been merged, and combinational and sequential observability cost estimates have been merged. It has been explained that these mixed cost estimates improve the heuristic steering during STPG. Finally, two pre-processes have been proposed to identify untestable faults: The first process identifies untestable faults, based on uninitalizable and/or unobservable signal lines (pre-process FUN). The second process uses a combinational TPG to identify combinational redundant faults.

A three-phase STPG has been described. Phase 1 considers the fault activation, and propagation in timeframe 1. Phase 2 considers the fault propagation in timeframe 2 and further until a PO has been reached. Phase 3 considers the state justification backward in time, until the current state covers the last state of the previously generated test sequence, or the 'all unknown' initial state. The STPG algorithm is based on the CTPG algorithm in DAT [vdL96], which in turn is based on the FAN algorithm [Fuj83], and incorporates techniques from HITEC [Nie91]. Many aspects of this CTPG algorithm have been modified to handle sequential (3-state) circuits: D-front propagation and dynamic dominators, six-valued objectives in the backtrace procedure, three valued decisions and fault dependent decisions, and bus-padding per timeframe. New techniques have been added to improve the performance of the STPG: Fault dependent cost estimates, complete and local fault-cones, the state restrictor to constrain the state space, and two techniques to perform some form of back-jumping. Eight STPG (search) strategies have been proposed to improve the performance for classes of sequential circuits.

To support the STPG, two static learning techniques have been proposed: To find global implications, a learning process has been proposed using the logic rules of constructive dilemma and contraposition, and recursive learning. Experimental results have shown an increased number of learned global implications, and/or reduced CPU costs compared to previous published methods. To find signal lines that are uninitializable to 0 and/or 1, a learning process has been proposed that tries to justify signal lines to value
0 and/or 1. If that fails, a new signal line which is uninitializable to 0 and/or 1 is identified.

Two additional target STPG methods have been proposed: The first one extends the current test sequence, by selecting additional targets making use of the fault-effects on the PPO state of the last generated timeframe. The STPG generates additional timeframes to propagate these fault-effects to a PO. The second one extends the current test sequence, after STPG for a primary target, using the unassigned PIs which are often available in the generated sequences. The STPG tries to assign these PIs to detect additional faults. Experimental results show that both methods increase the fault efficiency for a number of ISCAS'89 circuits. Especially, the performance of test generation for deep sequential circuits has been improved.

Finally, all strategies, methods, and techniques, which have been proposed in this chapter, have been combined into an ATPG process. This ATPG process is the basis for the proposals and experiments described in chapters 6, 7 and 8.
Chapter 6

ATPG based on Power-Set Logic

In Chapter 5, an ATPG system for sequential 3-state circuits has been presented. In this chapter, this ATPG system is extended with the Power-Set Logic (PSL) signal model. Using the power-set (i.e., the set of all possible subsets: \(0, 1, Z, 0\lor Z, 1\lor Z, 0\lor 1, 0\lor 1\lor Z\)) of the three basic values \(\{0, 1, Z\}\), a 128 valued (128V) signal model is constructed. Each value from the power-set is bitwise encoded, so each signal line has a set of still possible values from the power-set. Using this model, the three decision options \(0, 1,\) and \(Z(U)\) (see Section 4.1) are available on (P)PIs, to guarantee complete search during STPG. In addition, PSL provides efficient decision-making during STPG in terms of the number of backtracks, and avoids over-specified circuit states (see Chapter 4), because of its ability to express the exclusion of a certain value. Due to this ability, apart from decisions, also all re-decisions on (P)PIs during backtracking are made deterministically, in contrast to the random re-decisions made during STPG based on enumerative signal models. Furthermore, incorrect bus-conflict detection is avoided, due to the very high value resolution of PSL. Experimental results demonstrate the superiority of PSL over other signal models by reaching higher fault efficiencies for industrial as well as sequential ISCAS'89 circuits, combined with a large reduction of backtracking and computing time. The material described in this chapter has been published in [vdL98b].

This chapter is organized as follows: Section 6.1 presents the PSL signal model. Section 6.2 compares PSL with the SUV signal model (see Section 2.4) and the set-based signal model of MOSAIC. Section 6.3 describes the advantages of the PSL signal model for CTPG. In addition, modifications to the CTPG are proposed that are required for PSL based TPG in relation to the handling of 3-state circuits. Section 6.4 describes advantages of PSL based STPG. Section 6.5 discusses implementation issues. Section 6.6 presents experimental results for the ISCAS'89 [Brg89] and for industrial 3-state circuits. The results are compared to (C/S)TPG based on the 25V signal model [Kon95b]. Section 6.7 provides a summary and conclusions.
<table>
<thead>
<tr>
<th>Basic Set</th>
<th>Symbol</th>
<th>Logic value</th>
<th>Binary code</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>∅</td>
<td>–</td>
<td>0000000</td>
</tr>
<tr>
<td>{0}</td>
<td>0</td>
<td>0</td>
<td>0000001</td>
</tr>
<tr>
<td>{1}</td>
<td>1</td>
<td>1</td>
<td>0000010</td>
</tr>
<tr>
<td>{Z}</td>
<td>Z</td>
<td>Z</td>
<td>0001000</td>
</tr>
<tr>
<td>{0, 1}</td>
<td>0\lor 1</td>
<td>0\lor 1</td>
<td>0010000</td>
</tr>
<tr>
<td>{0, Z}</td>
<td>0\lor Z</td>
<td>0\lor Z</td>
<td>0100000</td>
</tr>
<tr>
<td>{1, Z}</td>
<td>1\lor Z</td>
<td>1\lor Z</td>
<td>1000000</td>
</tr>
<tr>
<td>{0, 1, Z}</td>
<td>0\lor 1\lor Z</td>
<td>0\lor 1\lor Z</td>
<td>1000000</td>
</tr>
</tbody>
</table>

Table 6.1: Encoding of PSL

6.1 PSL: A power-set based signal model

This section describes the Power-Set Logic (PSL) signal model. It has been shown in Sections 4.1 and 4.2 that the current signal models can lead to pessimistic bus-conflict detection, and/or are incomplete in search during TPG, and/or are inefficient in the decision-making process in terms of number of backtracks and in terms of causing over-specified circuit states during STPG. Therefore, PSL has been developed to perform TPG for 3-state and sequential circuits correctly and more efficiently.

Subsection 6.1.1 describes the PSL signal model and introduces the PSL signal value notation used throughout this chapter. Each signal line is assigned a default value, because not all possible values are useful to be assigned to a signal line; e.g., the output of an AND gate can only become value 0, or value 1, but not value Z. Subsection 6.1.2 explains the default value of the signal lines.

6.1.1 Description and notation of PSL

The PSL signal model uses the power-set (the set of all subsets) of the three basic values $BV = \{0, 1, Z\}$:

$$P(BV) = \{\{\}, \{0\}, \{1\}, \{Z\}, \{0, 1\}, \{0, Z\}, \{1, Z\}, \{0, 1, Z\}\}$$

$P(BV)$ consists of eight basic sets. Table 6.1 shows how the basic sets of PSL are encoded. Each basic set in set $P(BV)$ (column 1) corresponds to a symbol (column 2), a (disjunction of) logic value(s) (column 3), and a binary code (column 4). For example, basic set $\{Z\}$ with symbol $Z$, corresponds to logic value $Z$, and has binary code 0000100. The PSL value of each signal line is a set of basic sets, binary encoded, which denotes the still possible logic values on that line. A (disjunction of) logic value(s) can be assigned to a signal line if its corresponding bit (in the binary code) is still true (i.e., has value '1'). A signal value will be denoted as a set of symbols of Table 6.1, comma-separated and between normal braces to avoid confusion with the basic set notation; if the number of symbols in a set is one, the braces are omitted. In that case also the logic value can used instead of the symbol value. E.g., if a signal line has logic value 0, binary code 0000001, denoted as 0, is assigned to this line; if a line still can either become 0 or become Z, the
6.1. PSL: A power-set based signal model

binary code 0000101 (binary OR of 0000001 and 0000100), denoted as (0, Z), is assigned to this line. A signal line has a fixed value when exactly one bit of the binary code is assigned; i.e., it is fully specified. If multiple bits are assigned, the line is not fully specified (i.e., is X in traditional signal models). Hence, each signal line has a set of possible logic values, bitwise encoded into one computer word.

An important characteristic of set based logic signal models is the ability to express the notion that a value cannot be assumed (anymore). For example, consider a 3-state signal line with initial value (0,1,Z). Value 0 is decided by the TPG; however, this value causes a conflict, hence value 0 is removed from the signal line and excluded from the initial value of the 3-state signal line, resulting in value (1,Z). This characteristic is very important for an efficient decision making during TPG (see Section 6.3 and Section 6.4).

The eight basic sets are bitwise encoded for the fault-free, as well as for the faulty circuit. E.g., 0010101/1111110 expresses that the signal line can still be assigned logic values 0, Z, or 0\lor Z in the fault-free case, and all values except logic value 0 in the faulty case. The fault-free and the faulty part each may assume 128 different values, conceptually making PSL for TPG a 16384 valued signal model.

Values 0\lor 1 (01), 0\lor Z (0Z), 1\lor Z (1Z), and 0\lor 1\lor Z (01Z) are four different values, which have in common that they are fixed unknown values (value U in traditional signal models). They form a disjunction of the basic values 0, 1 or Z, but these basic values are not distinguishable. E.g., value 0\lor Z denotes a fixed value on a signal line, which is value 0 or value Z, but cannot be further specified.

The difference between the Boolean values 0 and 1, and 0\lor 1 can be made clear using the (allowed) logic values for a PI and PPI. A PI can either become value 0 or value 1 (symbol: (0,1)). A PPI can either become value 0, value 1 or value 0\lor 1 (symbol: (0,1,01)). Value 0\lor 1 can only occur on a signal line when it is driven by a PPI. It denotes a specified but unknown value for a PPI. As explained in Section 4.1, Page 57, this value is used during STPG to express the unknown initial (or power-up) state, to avoid over-specified states, and to guarantee complete search in STPG.

Figures 6.1a and b show the difference between 0, 1 and Z (symbol: (0,1,Z)), and 0\lor 1\lor Z: The latter value denotes a fixed unknown value; the output of the bus is fixed because all inputs values are fixed; the input values of the bus evaluate to an unknown output value (actually there is a bus-conflict); value 0\lor 1\lor Z is the disjunction of the values
0, 1 and Z, and these values are not distinguishable\textsuperscript{1}. The output of the bus in Figure 6.1b is not fixed, and still can become either 0, 1 or Z, depending on the values assigned to the switch which drives the bus; i.e., the output value of the bus can still be further specified.

Figures 6.1c and d show the difference between 0 and Z (symbol: (0,Z)), and 0\text\lor Z: Value 0\text\lor Z (see Figure 6.1c) denotes a fixed value (i.e., fully specified value); it has to be logic value 0 or Z. Value (0,Z) (see Figure 6.1d) denotes a (partially) unspecified value: The output of the switch still can become either fixed 0 or fixed Z.

6.1.2 Default values of signal lines

In PSL, each signal line initially has value \(P(BV)/P(BV)\); i.e., the fault-free and faulty part of the line can still assume all values of \(P(BV)\). However, usually not all logic values in \(P(BV)\) are possible on a signal line. E.g., the output of an AND gate can only have Boolean values, hence only values 0, 1 and 0\text\lor 1 are possible output values of an AND gate. A pre-process determines the default values; i.e., the possible set of logic values for all signal lines. A PI, not driving a 3-state element, can only be assigned value 0 or 1, hence its default value is (0,1). A PPI can become 0, 1 or 0\text\lor 1 (i.e., default value is (0, 1, 01)).

The pre-processes, described in Section 5.1.1 (can-be-Z and can-be-U calculation) and Section 5.1.3, Page 81 (pre-process FUN; uninitializable and unobservable signal lines), and the learning processes, described in Section 5.4, can make the default value of a signal more precise. E.g., the values Z, 0\text\lor Z, 1\text\lor Z, and 0\text\lor 1\text\lor Z are removed from the default value when a signal line cannot become Z (identified by the can-be-Z pre-process); i.e., the default value of such a line is (0,1,01). The fixed values on signal lines which have been identified by static learning are stored in the default value of these signal lines. Also, the information of uninitializable to 0/1 signal lines (identified by pre-process FUN and static learning) are stored in the default values. E.g., if a signal line is uninitializable to 0, then its default value is (1,01) (assuming a signal line driven by Boolean gates and PPIs).

Figure 6.2 shows a NAND gate. One input of this gate is driven by a FF, which is uninitializable to 1. Hence, the output of the NAND gate cannot become value 0. The default value of that signal line is (1,01). With appropriate default values, more implications can be performed in the circuits, which improves the performance of TPG

\textsuperscript{1}Although value Z is part of the disjunction value 0\text\lor 1\text\lor Z, the output of a conflicting bus cannot be high impedance (Z). In the case the a bus is conflicting, value 0\text\lor 1\text\lor Z is applied to express that the output of the bus is indeterminate; it can be any value between logic 0 and 1.
because conflicts are detected earlier, saving backtracks and improving fault efficiency (see Section 6.3 and Section 6.4).

6.2 PSL compared to other signal models

In this section, PSL is compared to the 5UV/25V signal model\(^2\), and the set-based signal model MOSAIC [Dar97] (see Section 4.2), to show the much higher value resolution of PSL which leads to faster and extended implications.

The main disadvantage of the 5UV signal model is the low value resolution on a signal line: A line has value \(X\), until the value is fully specified. But it is not possible to distinguish between a signal line which can become 1 and \(U\), and a line which can become 0, 1 and \(U\): Both lines have value \(X\). Figure 6.3a shows an example: The PPI cannot become 1; hence, the output of the AND-gate also cannot become 1. The 5UV signal model is not able to express this situation: The value of the PPI as well as the output of the AND-gate remains at \(X\) (see Figure 6.3a). However, PSL can express the situation described above: The PI can only become 0 or 1 (value \((0,1)\)), the PPI can only become 0 or \(\dot{0}\cdot1\) (value \((01,0)\)), resulting in an AND gate output value of \((01,0)\).

Figure 6.3b shows another example: The output of the bus remains \(X\), because the 5UV signal model cannot express that the output of the switch can only become 0 or \(Z\) (and needs the bus-patch (see Section 2.4) to correct the bus output to 0). PSL correctly evaluates the bus output to 0.

Value \(X\) in the 5UV signal model represents 120 different values in the PSL signal model; this illustrates the much higher resolution of the PSL signal model.

The low value resolution of the 5UV signal model is also due to value \(U\). In Section 6.1 has been described that value \(U\) represents four values in the PSL signal model \((\dot{0}\cdot1, \dot{0}\cdotZ, 1\cdotZ, \text{and } \dot{0}\cdot1\cdotZ)\); all four “\(U\)"-values can occur in the circuit. Figure 6.3c shows a switch output value \(U\) when the 5UV signal model is used, and \(\dot{0}\cdotZ\) when PSL is

\(^2\)Other enumerative signal models (as described in Section 4.2) have the same (or more) disadvantages as the 5UV signal model.
used. The 5UV signal model cannot express that the output value \( U \) of the switch 'means' \( 0VZ \). The bus output, in this figure, has value \( U \) when the 5UV signal model is used, and 0 when PSL is used, because value 0 and value \( 0VZ \) on the bus inputs evaluate to value 0 on the bus output.

In [vdL94b][Kon95b] the bus-patch was proposed, to solve this lack of resolution of the 5UV signal model as shown in Figures 6.3b and c, such that the output of the buses in Figures 6.3b and c will evaluate to 0 (see Section 2.4). However, the situation shown in Figure 6.3a is not solved by the 5UV signal model with bus-patch\(^3\). PSL does not require the bus-patch, which can be computing time expensive and makes PSL more elegant than the 5UV signal model.

The 16 valued, set-based signal model of MOSAIC, proposed by Dargelas [Dar97] (see Section 4.2), can handle the situation described in Figure 6.3b, but it incorrectly evaluates the output value of the bus in Figure 6.3c to \( U \) and erroneously reports a bus-conflict in that situation. The signal model of MOSAIC has a low value resolution with respect to value \( U \); similar to the 5UV signal model it cannot express that the output value \( U \) of the switch 'means' \( 0VZ \).

In [vdL94b] it has been mentioned that not all elements are associative under the 5UV signal model: For example, repeated table-lookup for a three-input bus can evaluate to incorrect output values. Figure 6.4a shows that the 5UV signal model is not associative: By applying repeated table-lookup, the bus output value evaluates to \( X \), while this should be \( U \). A (second bus-)patch is required as described in [vdL94b], to correct the output value of the bus to value \( U \). PSL is associative for all elements; the output value of the three-input bus becomes \( 0V1 (= U \) in 5UV) as shown in Figure 6.4b. This saves CPU time in comparison to the 5UV signal model, and allows for more simple, elegant code in the implementation (see Section 6.5).

To summarize, compared to other signal models, PSL allows for better performance in general because it:

\(^3\)Of course, the "bus-patch" can be extended so that it also can handle the situation described in Figure 6.3a.
6.3 PSL and TPG for combinational 3-state circuits

<table>
<thead>
<tr>
<th>Default value PI: ((0,1,Z))</th>
</tr>
</thead>
<tbody>
<tr>
<td>decision</td>
</tr>
<tr>
<td>a) 25V:</td>
</tr>
<tr>
<td>b) PSL:</td>
</tr>
</tbody>
</table>

![Figure 6.5: 25V CTPG compared to PSL CTPG backtracking mechanism](image)

- identifies more implications in the circuit, due to the higher signal value resolution.
- does not require the bus-patch to prevent incorrect bus-conflict detection, and
- is associative for all elements, hence does not require the ‘second bus-patch’.

6.3 PSL and TPG for combinational 3-state circuits

Based on PSL, a CTPG algorithm for 3-state circuits has been implemented. All techniques described in [vdL96] are applied in this CTPG algorithm. This enables a good comparison between 25V based CTPG (as proposed in [vdL96], etc.) and the new PSL based CTPG algorithm. In this section, we describe six advantages of PSL CTPG for 3-state circuits, compared to other CTPGs based on other signal models. Also, the switch-padding procedure is described, which is a procedure similar to bus-padding, and “pads” all switch outputs to values that guarantee that bus-conflicts cannot occur anymore during random padding (random padding\(^4\) is performed after a test has been generated). The six advantages of PSL applied in CTPG are:

1. **Improved exhaustive implication.** In Section 6.1 many examples have been given which show that logic implication based on PSL is more powerful than logic implication based on other signal models: The PSL values on the signal lines give much more information than the 25V signal values, due to the much higher value resolution of PSL.

2. **Improved backtrack mechanism.** PIs which can drive 3-state elements are allowed to become \(Z\). These PIs have three decision options during CTPG (see Section 4.1.2). If an initial decision on such a PI fails, then CTPG based on the 25V signal model (or other enumerative signal models) has to remake this decision to one of the alternative decisions (see row \(a\) in Figure 6.5). If this alternative decision also fails, then the decision has to be remade to the second alternative decision, to guarantee complete search in CTPG. Note, that often both alternative decisions do not contribute to the test. An important flaw in this backtracking mechanism is that the alternative decisions are taken randomly; i.e., if

\(^4\)i.e., random specifying not fully assigned values in order to increase the (“accidental”) fault coverage per test pattern.
CTPG backtrack remakes $0$ due to conflict to $1$ (or $Z$) this is not done because the $1$ (or $Z$) is known to contribute (to the test), but to reflect that it is not $0$. The ability of set based logics (e.g., PSL) to express the notion that a value cannot be assumed anymore, solves this.

The backtrack mechanism of PSL based CTPG is as follows: It remakes an initial decision on a PI (with three decision options) to the default value of this PI with the initial decision value excluded (see row $b$ in Figure 6.5). In the example of Figure 6.5: Value $0$ conflicts, hence $0 = \{0, 1, Z\} - \{0\} = \{1, Z\}$ is re-decided on the PI. This remade decision $(1, Z)$ may be later refined, to a fully specified value (i.e., to value $1$ or value $Z$). Figure 6.6 shows this way of decision-making, which guarantees complete search in CTPG. Note that this refinement of the (remade) decision is done deterministically (by the backtrace procedure of CTPG); i.e., it is known that value $1$ (or $Z$) is essential for the test (at this moment).

The advantage of this backtracking mechanism is a much more efficient decision-making process in terms of the number of backtracks, compared to 25V CTPG. Figure 6.7 illustrates this claim. Consider an untestable fault. To prove that the fault is untestable, the complete search space has to be (implicitly) traversed in CTPG. In 25V CTPG, for each made decision on a PI (assuming that these PIs have three decision options), two alternative decisions have to be tried to be complete in search. In addition, many deci-
6.3. PSL and TPG for combinational 3-state circuits

![Diagram 6.8: Improved multiple backtrace: Only input i1 can justify the output value](image)

**Figure 6.8:** Improved multiple backtrace: Only input i1 can justify the output value

![Diagram 6.9: Improved D-front and dynamic dominators determination](image)

**Figure 6.9:** Improved D-front and dynamic dominators determination.

sions may have been made after each of these alternative decisions (see Figure 6.7a). In PSL CTPG, only one alternative decision needs to be tried (and only if it is deemed useful by the backtrace procedure, this alternative decision is further refined) and if this decision also fails (possibly after many other decisions), the fault is proven untestable (see Figure 6.7b).

The ability of the PSL signal model to express that a signal line can *not* assume a certain value (e.g., value $\bar{0} = \{1, Z\}$) is very important to guarantee efficient decision-making, because random remaking of decisions to one of a list of alternative decisions is avoided (as occurs in 25V CTPG and other enumeratively based CTPGs).

3. **Improved multiple backtrace.** Consider the circuit (fragment) in Figure 6.8: On the output of the AND-gate value $\bar{0}$ is requested. The Multiple Backtrace (MB) in 25V CTPG has to make a choice between the inputs, because both inputs are candidate to justify the requested output value. However, the MB, based on PSL, determines that input $i2$ can never become $\bar{0}$, hence by implication input $i1$ has to be assigned $\bar{0}$ to justify output value $\bar{0}$.

4. **Improved D-front and dynamic dominators determination.** PSL CTPG removes an element from the D-front when that element cannot propagate the fault because the off-path inputs of that element cannot have a non-dominant value (but still are unspecified). 25V CTPG will keep that element in the D-front. Figure 6.9a shows an example of such a D-front element: The off-path input $i1$ cannot become $I$, hence the discrepant value on input $i2$ can never be propagated to the output of the AND-gate. PSL CTPG will discard this element from the D-front.
Figure 6.10: Two buses with non-fixed output values. However, only bus b has to be padded

Figure 6.9b shows a dynamic dominator example: The AND-gate is a dominator of fault $F$. Dynamic dominators determination with PSL detects that the off-path input of the dominator can not have a non-dominant value; i.e., fault $F$ cannot be propagated to the output of the dominator.

5. Only essential values on PIs. Over-specification of the test vector can occur due to remade decisions on PIs; these remade decisions often do not contribute to the test. In 25V CTPG, the remaking of decisions (on PIs) always results in the assignment of a fixed value to a PI (see row $a$ in Figure 6.5), hence these fixed PIs are part of the test, but do not necessarily contribute to the test. These (random chosen) alternative decisions on signal lines are performed to avoid that CTPG decides the same conflicting initial decisions again on these lines.

In PSL CTPG, over-specification of the test vector occurs less frequently when the PIs have three decision options. Remade decisions on these PIs usually are not a fixed value (see example of ‘improved backtrack mechanism’); i.e., the corresponding entries in the test pattern are kept at X. The remade decisions on these PIs do not contribute to the final test pattern, and are not essential to test the fault. Often, more PIs will remain at value $X$ (compared to CTPGs with other signal models), giving more ‘freedom’ for additional target faults during CTPG (see Section 5.6). This may increase the fault coverage per test pattern, resulting in more compact test sets.

6. Bus-padding. Bus-padding (see [vdL94b]) is performed only on buses which can become conflicting. Bus-padding is necessary after a test pattern has been generated for one or more faults, to avoid bus-conflicts. All buses, which still can cause a bus-conflict (i.e., those buses that do not have a fixed output value) are padded to non-conflicting input value combinations. Figure 6.10 shows two buses which will be bus-padded during 25V CTPG. However, only bus $b$ really has to be padded, because simultaneously driving 0 and 1 to this bus is still possible; value $01Z$ is still part of the output value set (when the output value of the bus becomes $01Z$, a bus-conflict is detected). The output value of bus $a$ is not fixed, but cannot become conflicting. PSL CTPG detects this situation and only performs bus-padding on buses which have $01Z$ in their output value set. Thus during PSL CTPG fewer buses have to be padded (compared to TPGs with other signal models).

In general, this improves obtained fault efficiencies and/or CPU times, and may result in more compact test sets.

Sometimes there are switches driving a bus, which have an output value assuring a con-
6.4. PSL and TPG for sequential circuits

Figure 6.11: Backward implication can not detect that some input value combinations are not possible

Conflict-free bus, however, this output value has not been justified by the input values of the switch; i.e., the output value can still change. Figure 6.11 shows an example. During CTPG, input a of the bus is assigned value 0. By implication, input b of the bus is assigned value (0, Z) (to avoid a bus-conflict). There are no values implied on the control and data input of the switch, due to the value on line b. However, input value combination ‘1, 1’ on the data and control input results in value 1 on line b, which is not allowed on that line; this value causes a bus-conflict. However, removing value 1 from the data and/or control line is not allowed: Value combinations ‘1, 0’ and ‘0, 1’ on the data and control line are allowed; i.e., value 1 is allowed on either the data or the control line. To avoid that value combination ‘1, 1’ is assigned on the switch inputs (by for instance, random padding), in PSL CTPG the 3-state switches have to be padded (switch-padding) to a fixed output value, to avoid bus-conflicts. This padding of the switch outputs is performed after bus-padding. In 25V CTPG this situation cannot occur. However, in 25V CTPG, the bus (in Figure 6.11), driven by the switch, would have to be padded. Hence, a similar CTPG effort is required for PSL CTPG and 25V TPG for this situation.

This small shortcoming of the PSL signal model can only be solved when a symbolic signal model is used. Such a model can express relations between signal lines. In the case of the switch in Figure 6.11, the relations between the inputs and output would be expressed in formulas, in such way that the 1, 1 input value combination on the inputs is excluded. However, TPGs based on a symbolic signal model are memory and CPU inefficient and (therefore) can only be applied for very small circuits.

Summarizing this section, we can conclude that TPG for combinational 3-state circuits with the PSL signal model is more powerful than TPG with the 25V signal model or other signal models. This results in lower CPU costs and/or higher fault efficiencies, as well as reduced test-set sizes.

6.4 PSL and TPG for sequential circuits

The STPG as described in Chapter 5 has been adapted for PSL. The adapted STPG has the same advantages as CTPG for 3-state circuits (based on PSL), as described in the Section 6.3. In addition, two advantages of STPG, based on PSL are described in this section.
1. Efficient decision-making during STPG. PSL STPG guarantees complete search because each PPI has three decision options: 0, 1, and 01. In addition, PSL STPG guarantees efficient decision-making in terms of the number of backtracks (see Section 6.3, Paragraph "Improved backtrace mechanism"). Also, (during STPG) all PPIs in a timeframe with value 0 or value 1 are always essential (i.e., these values are necessary for the test; no over-specified circuit states) for the test and have to be justified. This can be not true in 25V TPG: Figure 6.12 shows an example of an over-specified state, due to the PPI with the non-essential value 1, which has been assigned after backtracking (see re-decision in the circle). This over-specified PPI state is justified, which can result in a longer test sequence than required. PSL STPG assigns \( \bar{0} = (01, 1) \) on the PPI after backtracking. Because this value is not fixed (fully specified), STPG will not try to justify this PPI.

Note that each decision (-value) made by the backtrace procedure on a PPI always does contribute to the test (at this moment in STPG), and that all remade decisions (due to the backtrace procedure) on a PPI always do not contribute (see Figure 6.13). This is the underlying reason for the efficient decision-making of PSL STPG. All 0 and 1 values on PPIs are always due to decisions made by the backtrace procedure (and/or are mandatory assignments); i.e., these values have been determined to contribute to the test, they are not randomly assigned as enumerative logic based STPG has to do in order to guarantee completeness of search. For example, when after the backtrace procedure value 0 is decided on a PPI, then this value contributes to the test at this moment. If value 0 causes a conflict, the decision is remade to \( \bar{0} = \{1, 0\lor 1\} \); this value is not essential (does not
6.5. **PSL implementation aspects**

\[
\begin{array}{cccc}
\text{a} & \text{b} & \text{c} & \text{d} \\
\text{0} & \text{0} & \text{x} & \text{0} & \text{illegal state} \\
\text{0} & \text{0} & \text{x} & \text{x} & \text{current PPI state} \\
\end{array}
\]

Imply \( \bar{d} = \{01, \ 1\} \) on entry 'd' to avoid conflict

**Figure 6.14:** Inverse implication on entry ‘d’ of current PPI state when only one entry differs

Contribute) and will not be justified by STPG (because it is not a fixed value). Thereafter, the STPG may refine the remade decision on this PPI (see also Figure 6.13) to value 1 (value \( \vec{0}/1 \) can not be decided); this refinement is a decision made by the backtrace procedure and hence, does contribute to the test and have to be justified. If another conflict occurs, value 1 on the PPI is remade to value \( \vec{0}/1 ((01)) (\vec{1} = \{1, \ 0/1\} - \{1\} = \{0/1\}\); this value is also not essential and will not be justified. In 25V STPG it is possible that an essential value on a PPI has to be obtained via backtracking. For example, consider a decision value \( \vec{0} \) on a PPI. Assume that this value causes a conflict. The decision is remade to a randomly chosen alternative value, e.g., value \( U \). If value 1 on this PPI is essential for the test, then it has to be obtained via another backtrack on this PPI (remake value \( U \) to value 1). Therefore, 25V STPG is less efficient in terms of backtracks compared to PSL STPG.

2. **Effective utilization of illegal states.** PSL STPG results in a more effective utilization of the illegal states. When the current PPI state covers an IS, and the number of different entries between these states is one, then the inverse value of the entry of the IS, which differs from the current PPI state can be implied to avoid a conflict. Figure 6.14 shows an example: Value \( \{01, \ 1\} \) is implied on entry ‘d’ of the current PPI state to avoid a conflict with the illegal state. This is not possible with 25V STPG, because value 0 can be either \( I \) or \( U \). Both values cannot be implied at the same time in 25V, whereas in PSL, value \( \{01, 1\} \) precisely expresses the disjunction of the two values.

### 6.5 PSL implementation aspects

This section describes the implementation aspects of PSL. Each signal line can have 16K different signal values; therefore an efficient implementation of the element input and output evaluation functions is required to limit the CPU costs of those evaluation functions.

Similar to most other signal models [vdL96], the output value evaluations of the elements using PSL are table-driven. In addition, also all input value evaluations of the elements are table-driven. Other signal models usually apply a more complex procedural determination of input values during backward implication.

The output value table of a two-input element is denoted as \( E_o \), and the input value table of a two-input element as \( E_i \). The output value of an element is determined by (repeated) table-lookup in \( E_o \), using the current input values. Figure 6.15 shows a three input element \( E \), which can be described as two two-input elements \( E \). The output value
Out can be determined, using two table-lookups, as follows:

\[ \text{Out} = E_o(E_o(i1, i2), i3)) \]

An input value of an element can be determined using the current output value and the current other input values. E.g., the input value \( i3 \) can be determined using two table-lookups, as follows:

\[ i3 = E_i(E_o(i1, i2), \text{Out})) \]

This table-driven approach for input- and output evaluation simplifies the implication procedure: It does not have to distinguish anymore between the different element-types. In addition, this table-driven approach makes the application of PSL at least as fast as that of other signal models (e.g., the 5UV/25V signal model).

The tables \( E_o \) and \( E_i \) of an element are generated from the corresponding 5UV (see Section 2.4) output value tables. Except for the 1-input elements, only 128V input and output tables are generated and stored in memory. For 1-input elements also the 16KV input and output tables are generated; 16KV tables for two or more input elements require too much memory. To determine the 16KV input and/or output value of elements having more than 1 input, the 128V tables are used, while the fault-free and faulty part of the 16KV values are handled separately. After evaluation of the fault-free and faulty part of the 16KV input and/or output values, the resulting two 128V output and/or input values are merged into one 16KV value (by concatenating the two bit-patterns).

### 6.6 Experimental results

Combinational and sequential circuit TPGs employing PSL have been implemented in C++ within our ATPG system DAT. This section presents experimental results to demonstrate the effectiveness of PSL, by applying it to the combinational ISCAS’85 [Brg85] circuits, to full-scan industrial 3-state circuits, and to the sequential ISCAS’89 [Brg89] circuits. Two experiments have been performed on a PC with Pentium Pro 200Mhz processor and 256Mb main-memory under the Linux operating system.

The first experiment compares PSL TPG to 25V TPG for ISCAS’85 circuits and full-scan industrial 3-state circuits. Details on these circuits can be found in Appendix A and Appendix B. For each circuit, two ATPG runs have been performed; they were identical except for the signal model (i.e., they use the same strategies during TPG, and have equal time-limits). The first ATPG run uses the 25V signal model, and the second run uses the PSL signal model. The ATPG runs were focussed on speed; i.e., to get a high fault
### Table 6.2: Results of 25V and PSL ATPG on ISCAS'85 and industrial 3-state circuits

<table>
<thead>
<tr>
<th>Circuit</th>
<th>#nflts</th>
<th>#tst</th>
<th>#red</th>
<th>#pat</th>
<th>f.e.(%)</th>
<th>CPU(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>25V</td>
<td>PSL</td>
<td>25V</td>
<td>PSL</td>
<td>25V</td>
</tr>
<tr>
<td>c2670</td>
<td>2747</td>
<td>2630</td>
<td>2630</td>
<td>117</td>
<td>117</td>
<td>119</td>
</tr>
<tr>
<td>c3540</td>
<td>3428</td>
<td>3291</td>
<td>3291</td>
<td>137</td>
<td>137</td>
<td>175</td>
</tr>
<tr>
<td>c5315</td>
<td>5350</td>
<td>5291</td>
<td>5291</td>
<td>59</td>
<td>59</td>
<td>144</td>
</tr>
<tr>
<td>e6288</td>
<td>7744</td>
<td>7710</td>
<td>7710</td>
<td>34</td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>c7552</td>
<td>7550</td>
<td>7419</td>
<td>7419</td>
<td>131</td>
<td>131</td>
<td>229</td>
</tr>
<tr>
<td>SUM85</td>
<td>26819</td>
<td>26341</td>
<td>26341</td>
<td>478</td>
<td>478</td>
<td>701</td>
</tr>
<tr>
<td>p833</td>
<td>1546</td>
<td>818</td>
<td>819</td>
<td>721</td>
<td>727</td>
<td>38</td>
</tr>
<tr>
<td>p2133</td>
<td>2375</td>
<td>224</td>
<td>224</td>
<td>2151</td>
<td>2151</td>
<td>14</td>
</tr>
<tr>
<td>p2183</td>
<td>2258</td>
<td>2208</td>
<td>2228</td>
<td>26</td>
<td>27</td>
<td>247</td>
</tr>
<tr>
<td>p2736</td>
<td>2852</td>
<td>2473</td>
<td>2473</td>
<td>369</td>
<td>371</td>
<td>116</td>
</tr>
<tr>
<td>p7532</td>
<td>7898</td>
<td>5165</td>
<td>5102</td>
<td>2415</td>
<td>2753</td>
<td>362</td>
</tr>
<tr>
<td>p8429</td>
<td>9756</td>
<td>8115</td>
<td>8115</td>
<td>1641</td>
<td>1641</td>
<td>370</td>
</tr>
<tr>
<td>p24694</td>
<td>48888</td>
<td>48888</td>
<td>48888</td>
<td>0</td>
<td>0</td>
<td>56</td>
</tr>
<tr>
<td>p26603</td>
<td>31857</td>
<td>26166</td>
<td>26166</td>
<td>5674</td>
<td>5691</td>
<td>1352</td>
</tr>
<tr>
<td>p32118</td>
<td>34367</td>
<td>32105</td>
<td>32105</td>
<td>2252</td>
<td>2259</td>
<td>1389</td>
</tr>
<tr>
<td>p34592</td>
<td>32824</td>
<td>32799</td>
<td>32802</td>
<td>19</td>
<td>20</td>
<td>593</td>
</tr>
<tr>
<td>p34595</td>
<td>43334</td>
<td>33187</td>
<td>33178</td>
<td>10036</td>
<td>10097</td>
<td>751</td>
</tr>
<tr>
<td>p63578</td>
<td>67005</td>
<td>59353</td>
<td>59352</td>
<td>552</td>
<td>619</td>
<td>722</td>
</tr>
<tr>
<td>p65991</td>
<td>71929</td>
<td>63800</td>
<td>63822</td>
<td>7756</td>
<td>7978</td>
<td>1028</td>
</tr>
<tr>
<td>p132721</td>
<td>132924</td>
<td>127236</td>
<td>127253</td>
<td>5628</td>
<td>5671</td>
<td>723</td>
</tr>
<tr>
<td>p235318</td>
<td>225741</td>
<td>217774</td>
<td>217912</td>
<td>7596</td>
<td>7696</td>
<td>3025</td>
</tr>
</tbody>
</table>

#nflts: Total number of faults  
#tst: Number of faults detected  
#red: Number of redundant faults  
f.e.(%): Fault efficiency  
#pat: Number of test patterns generated

**efficiency as fast as possible at the expense of test set compactness. Before 25V/PSL ATPG starts, pseudo-random TPG [vdL96] is performed to remove the easily-detectable faults. After ATPG, static test set compaction is performed by means of reverse fault simulation. Table 6.2 shows the results.**

The results for the ISCAS'85 circuits demonstrate that PSL TPG is as fast as 25V TPG on combinational Boolean circuits. Except for some small variations in the number of test patterns, the results for both ATPG runs on ISCAS'85 circuits are equal (see also row 'SUM85' in the table). This is to be expected, because PSL TPG has no disadvantages over 25V TPG for combinational Boolean circuits; the advantage is software simplification.

The results for the industrial circuits demonstrate that PSL TPG compared to 25V TPG reaches higher fault efficiencies (e.g., for circuits p833, p7532, p132721), and is faster most of the time (e.g., for circuits p7532, p8429, p24694). Sometimes a higher fault efficiency results in slightly higher CPU costs (e.g., for circuits p65991, p235318). However, the total CPU costs of PSL TPG for the industrial circuits has been reduced by almost 15% (see row 'SUM'). The total fault efficiency of PSL TPG compared to
25V TPG (see row 'SUM') has been increased from 98.8% to 99.0%. Note that the fault efficiencies are very high for both TPGs. This means that the increase in fault efficiency of PSL TPG comes from the detection of the hard faults, which could not be successfully handled by 25V TPG. PSL TPG handles these faults, using the same backtrack limits as 25V TPG! If the number of redundant faults is relatively high, PSL TPG is always (much) faster (see circuits p7532, p8429, p34595); this indicates that PSL TPG requires less effort to prove faults redundant. This could be expected, considering that in PSL TPG much more mandatory implications can be performed and the decision-making process is more efficient (see Section 6.3).

The total number of generated test vectors by PSL TPG has been slightly increased (4%) compared to 25V TPG. In Section 6.3 we argued that the test set sizes would reduce because over-specification of the test vectors would less occur. However, the unassigned PIs after test generation are randomly padded; if the number of unassigned PIs increases in a generated test vector, because of less over-specification (in the case of PSL TPG), more PIs are randomly assigned a value, which can have a negative effect on the fault coverage per test vector. The application of additional target TPG (see Section 5.6) before random-padding, results in a larger increase of fault coverage per test vector for PSL TPG than for 25V TPG. Furthermore, the percentage of PIs (of the total PIs) which have three decision options in a combinational industrial circuit is usually low. Over-specification of the test vector by PIs which have two decision options also occurs in PSL TPG (in the same way as for 25V TPG).

The second experiment compares PSL ATPG to 25V ATPG for the sequential ISCAS'89 circuits. Similar to the first experiment, both ATPG runs are identical except for the signal model. Sequential circuit ATPG is performed as has been described in Section 5.7. Before sequential ATPG starts, first combinational TPG is performed to find (combinational) redundant faults. Then, two pre-processes are performed: To detect untestable faults (pre-process FUN, see Section 5.1.3, Page 81) and to determine global illegal states (GIS learning, see Section 7.2). When STPG has generated a test pattern, sequential fault simulation is performed to detect additional faults. Additional targets are selected based on fault-effects which have reached PPOs of the last generated timeframe (method A1, see Section 5.6). The time limit of the total ATPG run was set to 14400 seconds.

Table 6.3 shows the results of DAT-ATPG based on the 25V signal model and based on the PSL signal model. Table 7.4 in Section 7.6 shows the results from HITEC [Nie91], DUST [Gou93], and MOSAIC [Dar97]. Based on these results the following conclusions can be drawn:

- Compared to 25V STPG, PSL STPG performs much better: Higher fault efficiencies as well as lower CPU costs are obtained for almost all circuits. The total fault efficiency reached by PSL TPG increased by almost 8% (see row 'SUM.A'), while the total CPU costs have been reduced by 30%. Only 8 of the first 25 benchmark circuits do not reach 100% fault efficiency when PSL STPG is applied.

- Often the test set sizes have been reduced (e.g., for circuits s298, s344 and s400). This is as expected (see Section 6.4): Over-specified PPI states cannot occur in PSL
6.7 Summary and conclusions

STPG (and can occur in 25V TPG); these states can result in longer test sequences than required. However, sometimes the test set of a circuit generated by PSL TPG is larger than the test set generated by 25V TPG (e.g., for circuits s820, s713, s1488); but this is usually due to the increased fault efficiency of PSL TPG (i.e., more faults are detected).

- Compared to HITEC, DUST, and MOSAIC, the fault efficiencies of PSL STPG almost always are equal or better. In addition, the CPU costs of PSL STPG often are lower.

- The results for the largest ISCAS’89 circuits (s38417, s38584, s38584.1) are still poor. Often PSL TPG (as well as 25V TPG) cannot generate test sequences for most faults in these circuits. However, this is not due to deficiencies in PSL (therefore, we have excluded the results of these circuits from the summations in Table 6.3), but due to the state-of-the-art in sequential circuit TPG in general (see also results of HITEC, DUST, and MOSAIC in Table 7.4). Research has to be carried out to improve TPG for large sequential circuits.

6.7 Summary and conclusions

A novel signal model, called PSL, based on the power-set of the set of basic values \{0, 1, Z\} has been presented. PSL based TPG employs three valued decision options to guarantee completeness of search during STPG. In addition, PSL guarantees efficient decision-making during STPG and avoids over-specified circuit states, because of its ability to express the exclusion of a certain value; as a result, all specified values (value 0 and 1) on PPIs are actually assigned by the backtrace procedure (i.e., these values contribute to the test). It assures that only essential values on the PPIs are justified, reducing STPG effort and leading to more compact test sets. The pessimism in bus conflict detection of most other methods is absent from PSL, so it can be called exact within the realm of non-symbolic reasoning techniques. The value resolution is very high, such that much more implications can be performed in the circuit, compared to other signal models. Eight advantages of PSL based TPG over TPG based on enumerative and/or conventional set-based signal models (see Section 6.3 and Section 6.4) prove the superiority of PSL. Experimental results confirm this by reaching higher fault efficiencies for industrial circuits as well as sequential ISCAS’89 circuits, often combined with a large reduction of computing time.
<table>
<thead>
<tr>
<th>Circuit</th>
<th>#nflts</th>
<th>#st</th>
<th>#unt</th>
<th>seqtl</th>
<th>% f.e.</th>
<th>CPU(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>s27</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>0</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td>s208</td>
<td>215</td>
<td>137</td>
<td>137</td>
<td>78</td>
<td>78</td>
<td>136</td>
</tr>
<tr>
<td>s208.1</td>
<td>217</td>
<td>18</td>
<td>18</td>
<td>199</td>
<td>199</td>
<td>10</td>
</tr>
<tr>
<td>s298</td>
<td>308</td>
<td>265</td>
<td>265</td>
<td>43</td>
<td>43</td>
<td>253</td>
</tr>
<tr>
<td>s344</td>
<td>342</td>
<td>329</td>
<td>329</td>
<td>13</td>
<td>13</td>
<td>124</td>
</tr>
<tr>
<td>s349</td>
<td>350</td>
<td>335</td>
<td>335</td>
<td>15</td>
<td>15</td>
<td>123</td>
</tr>
<tr>
<td>s382</td>
<td>399</td>
<td>364</td>
<td>364</td>
<td>33</td>
<td>34</td>
<td>4012</td>
</tr>
<tr>
<td>s386</td>
<td>384</td>
<td>308</td>
<td>314</td>
<td>63</td>
<td>70</td>
<td>196</td>
</tr>
<tr>
<td>s400</td>
<td>424</td>
<td>381</td>
<td>382</td>
<td>40</td>
<td>41</td>
<td>4674</td>
</tr>
<tr>
<td>s420</td>
<td>430</td>
<td>179</td>
<td>179</td>
<td>251</td>
<td>251</td>
<td>176</td>
</tr>
<tr>
<td>s420.1</td>
<td>455</td>
<td>28</td>
<td>28</td>
<td>427</td>
<td>427</td>
<td>11</td>
</tr>
<tr>
<td>s444</td>
<td>474</td>
<td>424</td>
<td>424</td>
<td>48</td>
<td>49</td>
<td>3475</td>
</tr>
<tr>
<td>s510</td>
<td>564</td>
<td>0</td>
<td>0</td>
<td>564</td>
<td>564</td>
<td>0</td>
</tr>
<tr>
<td>s526</td>
<td>555</td>
<td>450</td>
<td>452</td>
<td>102</td>
<td>100</td>
<td>12572</td>
</tr>
<tr>
<td>s526n</td>
<td>553</td>
<td>450</td>
<td>452</td>
<td>100</td>
<td>99</td>
<td>12572</td>
</tr>
<tr>
<td>s641</td>
<td>467</td>
<td>401</td>
<td>404</td>
<td>63</td>
<td>63</td>
<td>140</td>
</tr>
<tr>
<td>s713</td>
<td>581</td>
<td>474</td>
<td>476</td>
<td>105</td>
<td>105</td>
<td>217</td>
</tr>
<tr>
<td>s820</td>
<td>850</td>
<td>662</td>
<td>744</td>
<td>14</td>
<td>15</td>
<td>400</td>
</tr>
<tr>
<td>s832</td>
<td>870</td>
<td>633</td>
<td>734</td>
<td>28</td>
<td>29</td>
<td>417</td>
</tr>
<tr>
<td>s838</td>
<td>857</td>
<td>254</td>
<td>254</td>
<td>603</td>
<td>603</td>
<td>172</td>
</tr>
<tr>
<td>s838.1</td>
<td>931</td>
<td>48</td>
<td>48</td>
<td>883</td>
<td>883</td>
<td>14</td>
</tr>
<tr>
<td>s935</td>
<td>1079</td>
<td>89</td>
<td>90</td>
<td>990</td>
<td>889</td>
<td>14</td>
</tr>
<tr>
<td>s1196</td>
<td>1242</td>
<td>1239</td>
<td>1239</td>
<td>3</td>
<td>3</td>
<td>378</td>
</tr>
<tr>
<td>s1238</td>
<td>1355</td>
<td>1283</td>
<td>1283</td>
<td>72</td>
<td>72</td>
<td>342</td>
</tr>
<tr>
<td>s1423</td>
<td>1515</td>
<td>148</td>
<td>543</td>
<td>14</td>
<td>20</td>
<td>11</td>
</tr>
<tr>
<td>s1488</td>
<td>1486</td>
<td>1202</td>
<td>1377</td>
<td>25</td>
<td>27</td>
<td>138</td>
</tr>
<tr>
<td>s1494</td>
<td>1506</td>
<td>1306</td>
<td>1364</td>
<td>37</td>
<td>39</td>
<td>323</td>
</tr>
<tr>
<td>s5378</td>
<td>4603</td>
<td>3079</td>
<td>3516</td>
<td>545</td>
<td>645</td>
<td>321</td>
</tr>
<tr>
<td>s9234</td>
<td>6927</td>
<td>18</td>
<td>18</td>
<td>6929</td>
<td>6909</td>
<td>5</td>
</tr>
<tr>
<td>s9234.1</td>
<td>6927</td>
<td>129</td>
<td>254</td>
<td>3151</td>
<td>3192</td>
<td>6</td>
</tr>
<tr>
<td>s13207</td>
<td>9815</td>
<td>509</td>
<td>646</td>
<td>8604</td>
<td>8688</td>
<td>22</td>
</tr>
<tr>
<td>s13207.1</td>
<td>9815</td>
<td>815</td>
<td>602</td>
<td>7849</td>
<td>9213</td>
<td>36</td>
</tr>
<tr>
<td>s15850</td>
<td>11725</td>
<td>85</td>
<td>85</td>
<td>11640</td>
<td>11640</td>
<td>7</td>
</tr>
<tr>
<td>s15850.1</td>
<td>11725</td>
<td>2872</td>
<td>3675</td>
<td>1129</td>
<td>1116</td>
<td>80</td>
</tr>
<tr>
<td>s35932</td>
<td>39094</td>
<td>28204</td>
<td>33896</td>
<td>3984</td>
<td>3984</td>
<td>68</td>
</tr>
</tbody>
</table>

SUM.A 119072 47150 54959 48644 50218 41464 25723 80.5 88.3 240138 169199

SUM.B 222858 58750 66398 57752 59419 41577 25832 52.3 56.5 283354 212421

#nflts: Total number of faults
#st: Number of faults detected
#unt: Number of untestable faults
seqtl: Test sequence length
% f.e.: Fault efficiency
CPU(s): CPU usage in seconds

Table 6.3: Results of 25V and PSL ATPG on sequential ISCAS'89 circuits
Chapter 7

Illegal state space identification

This chapter covers six techniques to expand the known Global Illegal State (GIS) space. As has been discussed in Section 4.1, reducing the size of the search space by identifying the GIS space is an effective method to improve the performance of the STPG, because the GIS space usually covers a large part of the search space.

Many GISes, which can be viewed as cubes defining the illegal state space, are implicitly ("accidentally") identified during STPG (see Section 5.2.7). Usually these GISes are over-specified; i.e., the GISes contain assigned entries which do not contribute to being illegal (i.e., they are not prime). Over-specified; i.e., non-prime GISes result in more GISes, spanning a smaller known GIS space (thus higher memory usage and higher CPU costs). Thus, eliminating over-specification is very attractive. Based on the fact that prime GISes are usually very sparse, we also propose a learning process to explicitly find prime GISes. To speed-up this process, we propose the use of a legal state cache, which stores all successfully justified states, to avoid useless justification repetitions of these states. A data-structure is proposed to reduce the memory usage of the (G)ISes (up to a factor of ten), and to increase the "handling-speed" of the (G)ISes. Experimental results for ISCAS'89 and some industrial circuits demonstrate the effectiveness of the proposed techniques in particular, and GIS learning in general. Parts of the material described in this chapter has been published [Kon97], or is submitted for publication [Kon99].

This chapter is organized as follows: Section 7.1 presents background information about GIS identification and optimization. Section 7.2 presents the pre-process GIS learning. Section 7.3 introduces the legal state cache to speed-up the justification process of potential GISes. Section 7.4 describes details on storage of (G)ISes. Section 7.5 introduces six techniques to expand the known GIS space. Section 7.6 presents the experimental results for ISCAS'89 and some industrial circuits, which have been compared to the results of HITEC [Nie91], DUST [Gou93], and MOSAIC [Dar97]. Section 7.7 summarizes and concludes this chapter.
7.1 Introduction to global illegal states

(G)ISes have been introduced in Section 5.2.7.1 to denote an illegal part of the state space (see also Section 4.1.3); i.e., when STPG enters that part of the state space, it can start backtracking.

In this section the concepts behind GIS identification and optimization are discussed in order to have a better understanding of the properties of GISes. The concepts are illustrated by means of Figure 7.1. It gives a representation of the state space of a circuit with 3 FFs. The complete state space of a circuit can be mapped onto a hyper-cube (assuming each state-bit to be completely independent, hence orthogonal to the others); in the 3 FF example this yields a 3-dimensional cube. Each corner of the hyper-cube corresponds to one completely specified state. Corners marked by a solid circle (●) are (completely specified) GISes. Assume that STPG found and stored GISes 100 and 101. The similarity between these GISes (i.e., they only differ in one entry) suggests that they may be parts of one larger GIS (subspace). The common part, 10X, is a potential GIS (PGIS). Hence one could use STPG's justification procedure to try to justify PGIS 10X, and (only) if this fails, the two GISes are removed from the GIS storage and replaced by GIS 10X, represented by the fat edge in the figure\(^1\). This was an example of GIS merging, which can also be illustrated by Figure 7.2: If two known GISes are relatively close to each other, they may be part of a larger GIS subspace. GIS merging has two advantages that are like a two-edged sword:

1. Less effective GISes are replaced by more effective GISes. A GIS is more effective if it covers a larger part of the state space hyper-cube (and thus is more sparsely assigned). A GIS is prime if it is maximal; i.e., no assigned entries of the GIS can be removed without making this GIS a legal state (i.e., cover partly the legal state space); the GIS cannot become more sparsely assigned.

2. The number of stored GISes is reduced. Having fewer GISes in storage has two advantages: It requires less memory, and the number of coverage checks to see if a certain state in STPG is already known to be a GIS decreases. The GISes in storage (i.e., they are stored in the GIS list) are irredundant if no GIS can be removed from the list without reducing the known GIS space.

\(^{1}\)Note: if states 100 and 101 are GISes, then state 10X is not necessarily a GIS; this has to be proven by justification of this state. If state 10X is a GIS, then also states 100 and 101 are GISes.
If all GISes are prime and the GIS list is irredundant, the memory usage of the GISes is minimal and no larger GIS space can be covered by these GISes.

Merging of GISes can also be tried when the difference is more than 1 bit: Consider that GISes 010 and 100 have been found. Consider also that there are more than 3 FFs, but that for this set of GISes all other FFs are unassigned, so these GISes can still be represented by a 3D cube. The fact that there are some GISes that have the same small subset of FFs assigned, means that these GISes are on a common hyper-plane, in the example a regular plane. By trying to justify the common hyper-plane as a PGIS, a better GIS (i.e., the GIS is "more prime") may be found, after which GISes 010 and 100 can be replaced by the single prime GIS XX0, which is twice as effective as the original 2 GISes but takes only half the storage space.

Alternatively, one may have found GIS 100, and consider that perhaps it is over-specified (i.e., not prime): Justification of 100 failed because 10X is unjustifiable. This can be tested by trying to justify state 10X, and if that fails, 10X is indeed a GIS, more effective than 100, and covering it, so GIS 10X is stored instead of 100. This is called GIS expansion. In Figure 7.2, GIS expansion can be viewed as trying to increase a known GIS to cover a larger part of the GIS subspace it is part of.

If a PGIS is tried, not generated from known GISes, but from 'random' trial or implicitly during test generation, we call this simply GIS identification.

The main target of GIS merging, GIS expansion, and GIS identification is to identify the complete GIS space, with prime and irredundant GISes; then the memory usage of the complete GIS space is minimal, resulting in increased STPG performance.

The main advantage of the concepts of GIS merging, GIS expansion, and GIS identification, over the existing approaches [Lon95] [Lia96] (see Section 4.3) is that the known GIS space is constantly increased, before and during STPG, facilitating the remainder of the test generation process. In addition, techniques based on these concepts use the justification procedure of STPG for the justification of PGISes; i.e., most techniques and heuristics used in STPG (see Chapter 5 and Chapter 6) are also useful for these techniques, making them very powerful. Also one justification procedure is needed for PGISes as well as for test generation, keeping the (program) code compact and elegant.

In the remainder of this chapter the justification procedure, which is phase 3 in the STPG (see Section 5.2) is called the Justifier, to avoid confusion between justification of
PGISes by the STPG and test generation by the STPG. Hence, PGISes are justified by the Justifier and test patterns are generated by the STPG.

Note that, although the concepts of merging, expansion and identification mostly apply to (fault independent) GISes, they also apply to the fault dependent ISes. All techniques proposed in this chapter can also be applied to the fault dependent ISes.

7.2 Static learning based on state justification to identify GISes

This section proposes GIS learning, a static learning technique to identify sparsely assigned GISes; i.e., GIS learning is a form of GIS identification.

Often the (prime) GISes of a circuit have very few assigned entries. E.g., the prime GISes of a BCD counter have only 2 assigned entries (1X1X and 11XX). A BCD counter is normally created with four flipflops, which often is only a very small part of the total number of flipflops in the circuit. For example, the ISCAS'89 circuits s382 and s526 each contain 21 flipflops and have three BCD counters: These circuits have prime GISes with only 2 assigned entries out of 21 (see also Figure 4.5 on Page 60). Thus often a large part of the illegal state space can be found when only the sparsely assigned states are tried to be justified.

GIS learning tries to find these sparsely assigned (very often prime) GISes, by means of justification of PGISes (using the Justifier). If justification fails, a GIS is found and stored in the GIS list. The PGISes are generated starting with a single assigned entry (either value 0 or 1 for each FF, one at a time), after which the learning process tries to justify these PGISes, one by one. If there is a PGIS (with a single assigned entry) that cannot be justified, then this means that the FF corresponding to the assigned entry of the PGIS cannot be initialized to the value of the assigned entry. Such an uninitializable FF yields the most effective GISes, because such a GIS reduces the considered search space in STPG by a factor 2; these GISes are always prime.

Thereafter, all states with two assigned entries (values 00, 01, 10 and 11 distributed over two FFs) are generated and tried, excluding the ones already covered by found GISes, and so on. Note that during justification of a generated PGIS, very often also other new GISes are 'accidentally' found, which were not explicitly generated as PGISes by the state generator. These GISes are found as described in Section 5.2.7.1, in the same way as they are 'accidentally' found in STPG itself.

To limit the CPU costs of this learning process, the PGISes are generated up to a maximum number of assignments, because the number of considered PGISes increases exponentially with the number of assigned entries. Table 7.1 shows the maximum number of assignments in a PGIS (column 2) as a function of the number of FFs (column 1). The values in column 2 are experimentally chosen based on the CPU costs of GIS learning. The third column shows the maximum number of generated PGISes ($maxP$, in worst case) based on the number of FFs (#FF, column 1) and the maximum number of assignments ($maxA$, column 2). The maximum number of generated PGISes can be de-
### Table 7.1: Maximum number of assignments in a PGIS and maximum number of PGISs to be generated in GIS learning

<table>
<thead>
<tr>
<th>$#FF$</th>
<th>Max. assignments in PGIS (maxA)</th>
<th>Max. generated PGISs (maxP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>≤ 12</td>
<td>$#FF$</td>
<td>531440 ($9^{12}$ - 1)</td>
</tr>
<tr>
<td>13</td>
<td>8</td>
<td>714194</td>
</tr>
<tr>
<td>14</td>
<td>7</td>
<td>714872</td>
</tr>
<tr>
<td>15</td>
<td>6</td>
<td>442346</td>
</tr>
<tr>
<td>16</td>
<td>6</td>
<td>686400</td>
</tr>
<tr>
<td>≥ 17, ≤ 21</td>
<td>5</td>
<td>≥ 242114, ≤ 758450</td>
</tr>
<tr>
<td>≥ 22, ≤ 30</td>
<td>4</td>
<td>≥ 130328, ≤ 472760</td>
</tr>
<tr>
<td>≥ 31, ≤ 60</td>
<td>3</td>
<td>≥ 37882, ≤ 280960</td>
</tr>
<tr>
<td>&gt; 60</td>
<td>2</td>
<td>$2^2 \cdot \left(\frac{#FF}{2}\right) + 2 \cdot #FF$</td>
</tr>
</tbody>
</table>

**Figure 7.3:** State A is justified by the state sequence $B \ldots F$

terminated as follows: $maxP = \sum_{i=1}^{maxA} 2^i \cdot \left(\frac{\#FF}{i}\right)$. Furthermore, this process can have a time limit, as well as the usual STPG limits on the number of backtracks. Of course, the PGISs which are covered by an earlier found GIS are not tried by the Justifier.

### 7.3 Legal state cache

This section introduces the legal state cache, which serves to avoid repeated justification of known legal states in order to speed-up the justification process of PGISs, for instance, during GIS learning.

Although often many PGISs (which have been generated by GIS learning) can be identified as being illegal, also many PGISs are justifiable. These justifiable PGISs, which are legal states, often have similar initialization sequences, or are justified by states which already have been proven justifiable earlier in the learning process. However, this information is not stored during GIS learning and the learning process continues justifying these PGISs, until the all-known state is reached. For example, Figure 7.3 shows a state sequence $A \ldots F$. State A has to be justified. To justify state A, a sequence of five states is generated. However, states $F$, $E$, $D$ and $C$ have already been proven justifiable earlier on by the GIS learning process, because they have fewer assignments than state A. The repeated justification of states $C$, $D$, $E$ and $F$ therefore would be useless. In order to prevent this and speed-up GIS learning (as well as GIS optimization techniques that are
proposed in Section 7.5), the Legal State cache (LS cache) is proposed to avoid useless repetitive justification of known legal states. Each time a PGIS has been justified, and hence has been identified to be a legal state, all states of the justification sequence are also identified as legal and are stored in the LS cache. If during the justification of a PGIS, the current state covers a state in the LS cache, justification can stop, since the current state is justifiable. The LS cache is called a "cache" because it has two properties of a cache: First, it is limited in size; i.e., the number of states which can be stored in the LS cache is limited. Second, only the states with a "high" hit-rate (i.e., those states which are often covered by states from the justification procedure) are stored. However, in contrast to a "real" cache which contains a subset of the contents of the "main" storage medium, all states with a few of no hits are destroyed. We use this kind of cache, in order to prevent that the identified legal states require too much memory.

The concept of the LS cache will be illustrated by means of an example. Consider an empty cache, which can store at most five states. Figure 7.4 shows 7 instances of the same cache A, B, C, ... G. Below these cache instances, the figure shows 6 PGISes (p1, p2, p3, ... p6) with their justification sequence. E.g., PGIS p1 (010) has the justification sequence 111 and 10X. The state sequence of Figure 7.3 is stored in the cache A. Cache instances B ... G show the contents of the cache after PGISes p1 ... p6 have been justified, respectively. The states which are shown bold in a cache instance are changed compared to the previous cache instance, or have been newly added into a cache instance.

A hit-rate value is associated with each stored legal state. The hit-rate value of a legal state is increased each time that a state being justified by the Justifier covers this legal state. The states in the cache are kept sorted in the order of the hit-rate values (highest first), and the number of assignments (highest first); when the hit-rate values of two states are equal, the state with the highest number of assignments is stored first, as shown in cache A. The learning process continues with PGIS p1 (010) and cache A (see Figure 7.4). State p1 does not cover any state in cache A. The search for a covered state
in the cache is performed top-down; i.e., first state $A_1$ and so on. States in the cache with fewer assignments than the current PGIS are skipped for comparison; e.g., state $p_1$ is not compared to states $A_3$, $A_4$ and $A_5$ because state $p_1$, having more assigned entries, could never cover any of these states. The Justifier continues justification: State $a$ justifies state $p_1$; state $a$ (111) also does not cover any state in cache $A$. State $b$ (10X) justifies state $a$; state $b$ covers state $A_2$ (101); i.e., PGIS $p_1$ can be justified and the Justifier stops with the justification process. The hit-rate value of $A_2$ is increased by one and moved upward in the cache (see cache $B$). Also, states $p_1$ and $a$ are stored in the cache (state $b$ covers a state in the cache, hence does not have to be stored). Because the cache is at its maximum size, states have to be removed. Two states ($p_1$ and $a$) have to be added, so the last (least effective/useful) two states of cache $A$ are removed ($A_4$ and $A_5$) and states $p_1$ and $a$ are inserted on their proper position (see cache $B$). Cache instances $C \ldots G$ show the cache after the corresponding PGISes $p_2 \ldots p_6$ have been justified.

It is clear that the highly assigned legal states with a high hit-rate value always "bubble" to the top of the stack. Legal states with a low number of assignments or a low hit-rate value ‘sink’ in the cache until they are removed (if the cache becomes full). Hence, the most effective legal states remain in the (top of the) cache, where they are also earlier found during matchings.

Legal states with a high number of assignments are more effective then states with a lower number of assignments, because the highly assigned states have more states which cover this legal state; (hence) these covering states are also legal. For example: Legal state 000 is covered by states $0XX$, $X0X$, $XX0$, $00X$, $0X0$, and $X00$; these six states are also legal. Legal state $00X$ is only covered by states $0XX$ and $X0X$. Thus, state 000 is much more effective than state 00X. To cover an as large as possible legal state space, the stored states in the cache should be highly assigned; as opposed to the illegal states, which cover a larger part of the illegal state space when they are more sparsely assigned.

The LS cache has been implemented within our ATPG system DAT. This paragraph presents experimental results to demonstrate the effectiveness of the LS cache, by performing GIS learning without and with LS cache on the ISCAS'89 circuits. The experiments have been performed on a PC with Pentium II 300 Mhz processor and 512Mb main-memory under the Linux operating system. The first experiment performed GIS learning without the LS cache. The second experiment performed GIS learning with the LS cache. The time limit for GIS learning was set to 2 hours.

Table 7.2 shows the results. To constrain the CPU costs of GIS learning, a maximum number of PGISes is generated for a circuit (see column 2); this value is determined using Table 7.1. Columns 3, 4, 5, and 6 show the results of GIS learning without LS cache and columns 7, 8, 9, and 10 show the results of GIS learning with LS cache. The number of PGISes which actually have been tried to be justified by the Justifier are shown in columns 3 and 7. Columns 4 and 8 give the maximum number of assignments ($A$) in the PGISes which have been successfully handled; i.e., no aborted Justifier runs, due to backtrack limits. This means that all PGISes having up to $A$ assignments have been tried, and hence all GISes having up to $A$ assignments have been identified. This is useful information for the Justifier as well as for the STPG, because no effort is required anymore to find GISes
<table>
<thead>
<tr>
<th>Circuit</th>
<th>Max. generated PGISes</th>
<th>Without LS cache</th>
<th>With LS cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>#PGIS tried 1)</td>
<td>CPU (s)</td>
</tr>
<tr>
<td>s208</td>
<td>242</td>
<td>242 8✓</td>
<td>4 1.73</td>
</tr>
<tr>
<td>s298</td>
<td>714872 620304 6</td>
<td>137 7200.0</td>
<td>7 137 1263.8</td>
</tr>
<tr>
<td>s344</td>
<td>442346 442346 6</td>
<td>1725 6913.4</td>
<td>6 1622 1838.4</td>
</tr>
<tr>
<td>s382</td>
<td>758450 8616 1</td>
<td>84 7200.8</td>
<td>5 291 3757.8</td>
</tr>
<tr>
<td>s386</td>
<td>728 728 6✓</td>
<td>10 8.41</td>
<td>728 6✓</td>
</tr>
<tr>
<td>s420</td>
<td>242 242 16✓</td>
<td>4 2.59</td>
<td>242 16✓</td>
</tr>
<tr>
<td>s526</td>
<td>758450 4987 1</td>
<td>47 7200.8</td>
<td>1 406 4696.0</td>
</tr>
<tr>
<td>s713</td>
<td>442586 442586 5</td>
<td>42 2326.9</td>
<td>442586 5 42</td>
</tr>
<tr>
<td>s820</td>
<td>242 242 1</td>
<td>6 653.48</td>
<td>242 5✓</td>
</tr>
<tr>
<td>s838</td>
<td>41728 41728 3</td>
<td>4 94.30</td>
<td>41728 3 4</td>
</tr>
<tr>
<td>s1196</td>
<td>330312 330312 5</td>
<td>42 2526.5</td>
<td>330312 5 699</td>
</tr>
<tr>
<td>s1488</td>
<td>728 600 –</td>
<td>7 7201.9</td>
<td>728 6✓</td>
</tr>
<tr>
<td>s5378</td>
<td>53138 53138 18</td>
<td>181 1470.7</td>
<td>53138 18 181</td>
</tr>
<tr>
<td>s13207</td>
<td>140450 1869 –</td>
<td>136 7207.3</td>
<td>1871 –</td>
</tr>
<tr>
<td>s35932</td>
<td>5971968 6980 –</td>
<td>381 7203.7</td>
<td>6980 –</td>
</tr>
<tr>
<td>SUM</td>
<td>9656482 1954920</td>
<td>3467 57213</td>
<td>3552915 4734</td>
</tr>
</tbody>
</table>

1) successfully tried PGISes up to #assignments
✓ : Complete GIS space has been identified
— : Aborted GIS justifications, even for PGISes with one assignment

Table 7.2: GIS learning without/with LS cache

having A or fewer assignments, saving computing time. The ✓ sign in columns 4 and 8 denote that all PGISes (see column 2) have been successfully handled, hence the complete GIS space has been identified, and thereby also its complement, the legal state space. The — sign in columns 4 and 8 indicate that there were aborted PGIS justification runs, even for PGISes with one assignment. Columns 5 and 9 give the number of identified GISes during the process, and column 6 and 10 give the CPU costs of GIS learning.

Based on these results, it is clear that applying the LS cache has resulted in a spectacular performance improvement: More GISes have been identified (or a larger GIS space has been identified, which is covered by fewer, more efficient (prime) GISes), and much more PGISes have been handled by the Justifier and/or the GIS learning computing costs have been significantly reduced.

### 7.4 Storage of (G)ISes

The subject of this section is storage of (G)ISes, because memory usage of the ISes, the access time to the ISes, and comparison speed between two states, are very important for the performance of the STPG. During STPG, often many ISes have to be stored; i.e., the memory usage can be (very) high. Hence, also many ISes have to be compared to a huge number of reached circuit states, to detect that illegal parts of the search space are entered. Hence, comparing the PPI or PPO state with the ISes have to be performed as fast as possible; i.e., a low access time to the ISes and a high comparison speed between
7.4. Storage of (G)ISes

Figure 7.5: a) FF-groups based on independent GISes; b) new FF-group due to GIS C, extension of a FF-group due to GIS D, and merging of FF-groups due to GIS E

ISes and circuit states are necessary.

In this section the concept of FF-groups (Section 7.4.1) and the concept of compressed (state) vectors (C-vectors) (Section 7.4.2) are proposed to improve these storage and handling requirements of ISes.

7.4.1 Groups of flipflops (FF-groups)

An FF-group is a group of FFs, and one or more GISes are associated to an FF-group. The FFs within one FF-group have the property that the GISes associated to this FF-group are dependent of each other in a way that they have common assigned entries. GISes associated to different FF-groups are independent of each other. The concept of FF-groups is based on the fact that GISes, in particular the sparse GISes, often are independent of each other; i.e., they do not have common assigned entries. The purpose of FF-groups is to accelerate comparison between two states and to reduce the number of states which have to be compared, saving STPG computing time.

Figure 7.5a shows two GISes A and B. When GIS A is stored in the GIS list, FF-group 1 is defined, consisting of the FFs of the assigned part of GIS A: FF 1 and 2. Storage of GIS B results in FF-group 2, because GIS B is independent of GIS A (the entries in GIS B corresponding to the FFs in group 1 are all X). FF-group 2 is formed by FFs 4 and 5, and GIS B is associated to that group. Note that an FF can be in only one FF-group and that if two GISes have one or more common assigned entries, then a FF-group is formed which consists of all FFs corresponding to all assigned entries of these two GISes.

During STPG runs, new GISes are stored in the GIS list, possibly resulting in the following changes in the FF-groups (see Figure 7.5b):

- A new FF-group can be formed. E.g., when GIS C is stored, a new FF-group is formed because the FF corresponding to the assigned entry of GIS C is not part of any FF-group.
- An FF-group can be extended with an FF. E.g., when GIS D is stored in the GIS list, FF-group 1 is extended with FF 3.
- Two or more FF-groups can merge into one FF-group, because a GIS is stored in the list, that covers those FF-groups. E.g., when GIS E is stored, group 1 and 2 have to be merged because GIS E has assigned entries in both groups.
The use of FF-groups has two advantages, which save STPG computing time; if during STPG an FF is assigned a value:

1. Only the GISes which are associated to the FF-group of that FF have to be compared. E.g., if FF 1 is assigned a value, only GIS A has to be checked (see Figure 7.5a).

2. Within one group, only the FFs from that group have to be compared to the corresponding entries of the GISes associated to that group. E.g., consider the two FF-groups in Figure 7.5a: Only assignments on FF 1 and 2 have to be compared to GIS A; assignments on FF 4 and 5 with GIS B. If FF 3 is assigned a value, no comparison is needed because this FF is not part of any already existing group.

### 7.4.2 Compressed vectors

A compressed (state) vector (cvector) is a vector of values, which forms a circuit state, whereby each value (the state of one FF) consists of three bits, instead of an integer (usually 32 bits) in the case of a regular vector. Each bit in such a value corresponds to one of the basic values 0 (binary code: 001), 1 (binary code: 010) and U (binary code: 100). In this way each value of a cvector is a set of basic values; e.g., binary code 011 represents the set of basic values 0 and 1 for a FF state value. Because an integer usually consists of 32 bits, within one integer ten 3-bit values (fault independent) or five 6-bit values (fault dependent; each fault dependent value actually consists of two 3-bit values, namely the fault-free value and the faulty value) can be stored. This results in compression ratios of 10 or 5 respectively, in comparison to vectors with values stored per integer.

An advantage of using set-based values in the cveectors is that, for example, states 0 X (binary code: 001 111) and 1 X (binary code: 010 111) are merged into a single state 0 v1 X (binary code: 011 111), resulting in further reduced storage requirements.

Compressed vectors (cveectors) improve the access-time, the comparison speed and the storage requirements of ISes. It is clear that cveectors reduce storage requirements (a state value consists of 3 or 6 bits instead of 32 bits), while, in addition, the comparison speed between ISes and between the current PPI/PPO state and ISes is improved: The values of the cveectors are compared integer-wise, so 10 or 5 values are compared in one integer-level step. Only when the exact FF index must be known in a value of the two cveectors (for instance, to identify the entry which has a difference with the current PPI state), a comparison at bit level of one integer value, which consist of ten 3-bit or five 6-bit values, one at the time, is necessary.

### 7.5 Expanding the known GIS space

In this section, six techniques are described which are based on the concepts of GIS identification, expansion, and merging (see Section 7.1), to expand the known GIS space, using the list of known GISes (GIS list). Often, the GISes in this list (those that were detected during STPG) are over-specified; i.e., one or more entries of these states do not contribute to the actual reason of their unjustifiability. Figure 7.6 shows two timeframes.
Figure 7.6: PPIstate1 is illegal. The assignment on PPI 3 is over-specified

The PPI state of timeframe 1 (PPIstate1) is unjustifiable. The assignment on PPI 3 is over-specified; i.e., this assignment is not the actual reason for the state being illegal. PPIstate1 is also illegal without the assignment on PPI 3. Therefore, the assignment on PPI 3 should be removed from this IS, to increase its effectiveness.

Determining new GISes using the list of known GISes has two important advantages (by applying the concepts of GIS expansion, merging and identification):

1. The new GISes usually have fewer assigned entries than the original GISes. Hence, the new GISes cover a larger part of the illegal state space (the new GISes are usually ‘more prime’ than the original GISes). E.g., GIS 0X covers GISes 00 and 01, while GIS 00 covers only itself.

2. Because the new GISes are derived from the original GISes, the new GISes cover one or more of the original GISes. These covered (and now redundant) states are removed from the GIS list, often resulting in a large reduction of the number of GISes in the list. This saves CPU time during STPG because fewer GISes have to be checked with the current state on the PPIs and saves memory.

The six techniques proposed to generate better GISes (i.e., GISes which expand the known GIS space) from the existing ones, are described in the following six paragraphs and are called:

\( T1 \): Expansion of single GISes by unassigning single entries.

\( T2 \): PGIS generation from two similar GISes by finding common fragments.

\( T3 \): PGIS generation from ranges of similar GISes by finding common fragments.

\( T4 \): Direct expansion, an improved version of \( T1 \).

\( T5 \): Generation of multiple PGISes from one GIS, to avoid merging of FF-groups.

\( T6 \): Generation of PGISes from existing groups of similar GISes to find new GISes for closely related FFs, such as in BCD counters, etc.

All six techniques generate Potential Global Illegal States (PGISes), which may become GISes after justification failure (by the Justifier). E.g., when states 00 and 01 are GISes
Expand_illegal_state( aState )
{
    unassigned=0
    finished=0
    while ( not(finished) ) {
        finished=1
        aborts=0
        for all assigned entries i of aState {
            a: unassign entry i of aState
                justify_result = Justifier( aState )
            b: if ( justify_result == SUCCESS )
                restore value of entry i
            c: else if ( justify_result == ABORTED ) {
                aborts=1
                restore value of entry i
            }
            d: else { //current aState is unjustifiable
                unassigned=1
                if ( aborts==1 ) finished=0
            }
        }
    }
    if ( unassigned==1 )
    e: remove all states from the GIS list covered by aState
        if ( aborts==0 )
    f: aState is marked 'prime'
}

Figure 7.7: Expand a single state by unassigning entries

and state 0X is generated as a PGIS, then this PGIS becomes a GIS only if justification
fails for this PGIS².

T1) PGIS: Unassign single entries of a GIS. Figure 7.7 shows the procedure for unassigning single entries of GISes. In [Gou93] a similar, but slightly less effective technique
was introduced. For each IS in the GIS list (until a time limit has been reached), procedure
Expand_illegal_state³ is performed: Assigned entries of the IS are unassigned (i.e., set to
X; line a in Figure 7.7), one at the time, and the Justifier tries to justify this modified state
(PGIS). This leads to one of the three next steps:

1. If the PGIS can be justified, then it is not a GIS, and the value of entry i is restored
   (see Figure 7.7 line b). The procedure continues with the next assigned entry of the
current GIS.

²Another example to illustrate the necessity of justification failure of PGISes (which are generated based
on GIS merging), before the PGIS is stored as a GIS: Assume the GISes 0 and 1. The merged state of these
two GISes is X. This merged state is not illegal because this is the initial (power-up) state of the circuit.
³In [Kon97] this procedure has been called Minimize_illegal_state; The "minimization" refers to reducing
the number of assigned entries in an IS.
7.5. Expanding the known GIS space

\[
\begin{array}{cccccc}
2 \text{ GISes} & & & & & \\
X & X & 0 & 0 & 0 & 1 \quad 1 \quad X \\
X & 0 & 0 & 0 & 0 & 0 \quad X \quad X
\end{array}
\}

\[
X & X & 0 & 0 & 0 & 0 \quad X \quad X \quad X
\]

Figure 7.8: Determine PGIS from the common fraction of 2 GISes: Set the differences between the 2 GISes to X in the PGIS

2. If the PGIS cannot be justified due to time limits; i.e., justification of the PGIS is aborted, then the value of entry \(i\) is restored and a flag \textit{aborts} is set (see Figure 7.7 line \(e\)).

3. If the PGIS is proven unjustifiable, then a new GIS has been detected. Entry \(i\) is kept unassigned. If any previous Justifier runs for the current GIS is aborted (i.e., flag \textit{aborts} is set), then the loop over all assigned entries is restarted after the current loop has been finished (see Figure 7.7 line \(d\)). This is done because often an aborted justification attempt for a PGIS with assigned entry \(i\) is not aborted in the next loop if one or more entries (including entry \(i\)) are kept unassigned in this loop. The possibility to restart the loop, when aborted justification attempts were detected, is the improvement compared to the technique proposed by [Gou93].

If one or more entries are kept unassigned, a new GIS has been detected and added to the GIS list.

All states covered by this new GIS are removed from the GIS list (see Figure 7.7 line \(e\)). If all over-specified assignments of a GIS are removed (i.e., the GIS is prime), the GIS is marked \textit{prime} so that it will not be a target anymore for expansion.

A process, called Expand-Iillegal-States (EIS), executes Technique T1 \textit{Expand.illegal.state} on the ISes in the GIS list. EIS is performed between two ATPG stages. The number of GISes in the GIS list can often be very large after an ATPG stage has been finished. By using this process, more effective GISes (covering a larger GIS space; i.e., they have less assignments than the original GISes) are determined, often covering a large number of the already known GISes, which are removed from the list.

**T2) PGIS: Common assigned fraction of two GISes.** Figure 7.8 shows two GISes which differ in three entries. A new PGIS is created from the common assigned fraction of the two GISes, with the different entries set to X. The Justifier tries to justify this PGIS. If the PGIS is unjustifiable, then a new GIS is found, and (at least) the original two GISes are removed from the GIS list.

Process Check-Iillegal-States (CIS) applies this technique (see Figure 7.9). This process compares all states in the GIS list with each other. If one state covers another state, the last (covered) state is removed from the list (see Figure 7.9 line \(a\)). Else, if the number of differences between two states is 3 or less (this value was experimentally chosen), a PGIS is created (as shown in Figure 7.8) with the different entries set to X, and stored in the list \textit{POT.ISL} (see Figure 7.9 line \(b\)).
CIS( GIS list )
{
    for all states $S_i \in GIS\ list$
        for all states $S_j \in GIS\ list; j > i$
            a: if $(S_i \subseteq S_j)$ then
                remove( $S_j$ )
            else if $(S_j \subset S_i)$ then
                remove( $S_i$ )
            b: else if (ndiffs($S_i, S_j$) $\leq$ 3) {
                create PGIS from common fraction of $S_i$ and $S_j$
                (see Figure 7.8)
                store PGIS in list POT.ISL
            }
    for all states $S_p \in POT.ISL$
        c: if (justify($S_p$) == FAILED) {
            store $S_p$ in GIS list
            remove all vectors $S_i \in GIS\ list | S_p \subseteq S_i$
            remove all vectors $S_i \in POT.ISL | S_p \subset S_i$
        }
    }

Figure 7.9: Process CIS: remove covered states, determine PGISes and try to justify these states

After these comparisons, the states in POT.ISL are tried to be justified. If the Justifier proves a state from the POT.ISL to be unjustifiable (see Figure 7.9 line c), then the state is stored in the GIS list, and all states in the GIS list as well as POT.ISL which are covered by this new GIS are removed.

The process CIS is performed during an STPG run, each time when the number of vectors in the GIS list modulo a predefined limit (usually a value between 50 and 100) has become 0. Care is taken that no pair of GISes is considered in CIS that has been considered in previous CIS runs. Process CIS is also performed on the other IS lists of the StateRestrictor (USL, FISL, OUSL and OISL; see Section 5.2.7.1) to remove these states covered by other states in the list. However, technique T2 is only performed on the states in the GIS list, because the states in the other lists are temporary (however, technique T2 (as well as other techniques proposed in this section) can be applied to these fault-dependent state lists, but to save on CPU costs, the techniques are only applied to the fault-independent GIS list).

T3) PGIS: Common assigned fraction of a range of GISes. This technique detects common (assigned) fractions in ranges of successively stored GISes. The common fraction (the entries that are the same for most of the range) has been decided/implied by the STPG to detect the fault-under-test, but is found to be unjustifiable later on in the STPG-
7.5. Expanding the known GIS space

process. However, after the common fraction was implied (and before that fraction is being justified), often more decisions are implied (on the PPIs) to test the fault. These later decisions are not the reason this state is unjustifiable, resulting in (successively stored) over-specified GISes (with a common fraction).

Figure 7.10a shows five GISes which have been determined during STPG for a fault. The reason that these five states are declared illegal by STPG is probably due to the common fraction of these five states: State X1X01XX. Figure 7.10b shows how the common fraction is detected. An array of integers, called common_fraction.array (cf), is updated each time a GIS is stored in the list by comparing the GIS to the previously added GIS. For each entry i of the GIS (GIS[i]) the corresponding entry i in the common_fraction.array (cf[i]) is updated according to the following three rules:

Rule 1. If GIS[i] = X then cf[i] = 0

Rule 2. If GIS[i] = b ∧ prev.GIS[i] = b (b ∈ {0, 1}) then cf[i] = cf[i] + 1

Rule 3. If GIS[i] = b ∧ prev.GIS[i] ≠ b (b ∈ {0, 1}) then cf[i] = 1

A PGIS is generated when the formula F

\[
\sum_{i=1}^{\#FFS} (cf[i] \geq MinVal) \geq cfLimit
\]

becomes true. Here MinVal = (0.75 × Max(cf)); Max(cf) is the maximum value of all entries of the array cf; value 0.75 has been experimentally chosen (its goal is to provide some margin in the selection of entries that establish the common fraction; i.e., "select high entries" rather than "the highest entries"); cf_limit denotes a predefined limit, which defines the minimal number of entries in cf with a value larger than or equal to MinVal. During STPG for a fault, cf is updated each time that a GIS is stored, as described above. If STPG has been finished for the fault, formula F is evaluated. If it becomes true, a PGIS is generated: The PGIS is a copy of the last stored GIS, with those entries unassigned for which the corresponding entry in cf has a value less than MinVal (the assigned entries of the last stored GIS which corresponding values in cf are larger or equal than MinVal form a common fraction, which is probably unjustifiable). For example, assume that the GISes shown in Figure 7.10 are from STPG for a single fault. Before STPG starts with the next fault, F is evaluated. Assume cf_limit = 3, than
Expand_and_Store_State( aState )
{
    Store_state in GIS list( aState )
    if ( nr_of_assignments( aState ) >= min_nassignments ) {
        Store in TO_EXPAND list( aState )
        if ( not State-expander is busy ) {
            Mark State-expander is busy
            while ( list TO_EXPAND is not empty ) {
                get a GIS from list TO_EXPAND
                Expandillegal_state( GIS //technique T1
            }
            Unmark State-expander is busy
            Remove covered GISes from the list
        }
    }
}

Figure 7.11: Store and direct expansion of GISes

\[ \sum_{i=1}^{7} (cf[i] \geq MinVal) \geq 3; MinVal = (0.75 \times Max(cf) = 0.75 \times 5 = 3.75) \text{ is true; i.e., the number of entries in } cf \text{ with value larger than } 3.75 \text{ is larger than } 3, \text{ and a PGIS is generated. The entries of the PGIS, corresponding to the entries in } cf \text{ with a value larger than } MinVal, \text{ are assigned the value of the corresponding entries of the last added GIS, resulting in: PGIS = X1X01XX. If justification fails for this PGIS (i.e., a new GIS is found), then Expandillegal_state (T1) is performed on this new GIS in order to attempt to unassign even more entries. All states covered by the resulting new GIS are removed from the GIS list.}

The common_fraction.array cf is always reset before STPG starts with a new fault. Hence, this procedure only detects common fractions in ranges of GISes, which are close together in the GIS list; i.e., detection of PGISes depends on the location of an IS in the GIS list. However, it is clear that the probability of having a common assigned fraction in closely together stored GISes (they are stored during STPG for one fault) is higher than between GISes stored for different faults.

T4: Direct expansion of a new GIS. Often GISes are stored in the GIS list, which would not be stored when the already stored GISes had been expanded using T1, because then the new GISes would have been covered by these expanded GISes. However, process EIS, which applies technique T1 on the ISes in the GIS list, is performed between the ATPG states and not during an ATPG state immediately when new GISes are added in the list.

Technique T4 solves this disadvantage of process EIS: Using a very low backtrack limit (16 or 32), each new GIS that has at least a predefined minimum of assigned entries, is expanded by T1 immediately after this GIS is stored in the GIS list. Hence, as soon
as a GIS is stored, somewhere during the ATPG process (e.g., during STPG for a fault or during the learning processes to find GISes), the GIS is expanded, removing over-specified entries as much as possible. Figure 7.11 shows the complete procedure to store and expand a new GIS. Each new GIS is always stored in the GIS list (line a in the Figure), and also stored in a list called TO.EXPAND if it has at least some predefined number of assigned entries (min.nassignments, line b). This list is sorted on the number of assigned entries of the GISes. When the State-expander (line d-f) is not busy with expanding (line c), Expand_illegal_state (technique T1) is performed on the first GIS in the list TO.EXPAND. During this expansion process, new GISes can be identified, which are also stored in the GIS list, and depending on the number of assigned entries also in list TO.EXPAND. The covered GISes are removed after the State-expander has finished (line g). The experimental results (Section 7.6) demonstrate that this technique is extremely useful for many circuits.

**T5: PGISes to preserve existing FF-groups.** In Section 7.4.1, FF-groups have been introduced to accelerate comparison between two states and to reduce the number of states which have to be compared. However, during STPG GISes can be identified which can result in that two or more FF-groups have to be merged into one larger FF-group. Figure 7.12 shows an example: Addition of GIS C in the GIS list results in the (necessary) merge of FF-group 1 and FF-group 2 because GISes A, B, and C have common assigned entries; i.e., these three GISes have to be associated to the same FF-group. Two disadvantages of merging FF-groups can be mentioned:

1. It increases the number of GISes which have to be compared to a circuit state, because all GISes associated to each of the FF-groups before the merge are now associated to the merged FF-group. For example: Assume that FF 1 (see Figure 7.12) of the circuit state is assigned a value. Before the merge of FF-groups 1 and 2, only GIS A has to be compared to the circuit state, and GIS B is not compared. After the merge, in addition to GIS A, GIS B and GIS C have to be compared to the circuit state.

2. It increases the comparison time per GIS, associated to the merged FF-group, because more FFs are part of the (merged) FF-group. For example: Assume that FF 1 of the circuit state is assigned a value. Before the FF-group merge, only entry 1 and 2 of GIS A (FF 1 and FF 2 are part of FF-group 1) are compared to the circuit state.
After the merge, in addition the entry 1 and 2, also all other entries of GIS A have to be checked with the circuit state, because FF-group 1 is merged with FF-group 2 and is extended with FF 3 and FF 6.

It is clear that avoiding merging of groups is very desirable. Technique T5 tries to avoid group merging, by generating PGISes for each group which has to be merged. It uses the GIS which caused the group merge, by copying the assigned entries for one FF-group of the GIS and setting the remaining entries unassigned. E.g., when GIS C (see Figure 7.12) is stored in the list, FF-groups 1 and 2 are merged. PGIS 1 is generated using the assigned part for FF-group 1 (0X0XXX) and keeping the remaining entries unassigned. PGIS 2 is generated using the assigned part for group 2 (XXX1X1). If any of the two PGISes cannot be justified then at least one new GIS (PGIS 1 and/or PGIS 2) is stored in the GIS list, which cover GIS C. GIS C is removed from the GIS list, resulting in preservation of FF-groups 1 and 2. Especially for circuits with a large number of FFs and/or very sparsely assigned GISes, this technique is successful. Note: How the groups are defined is dependent on the order in which the GISes are identified and stored in the list. E.g., if GIS C has been identified first, then only one group would have been defined and this technique could not have been applied.

T6: PGISes by enumeration of the assigned part of a GIS. This technique generates PGISes from existing groups of similar GISes to find new GISes for closely related FFs, such as in BCD-counters, etc. Two reasons motivate T6:

1. Before an ATPG stage starts, GIS-learning is performed to find (sparsely assigned) GISes. For this process, depending on the number of FFs in the circuit, a state-generator generates, PGISes with one, two, or more assigned entries, and these states are tried to be justified. Many GISes can be found in this way. However, for circuits that contain many FFs, only PGISes are generated with one assigned entry, which are often justifiable. It would take too much computing time for these circuits to generate and justify PGISes with two or more assignments.

2. Often, closely related FFs can be found in the circuit, such as the FFs of a BCD counter or an one-hot code. Often one or more states formed by these FFs are GISes. E.g., a BCD counter has 6 GISes (1010 up to 1111); for an one-hot code, all states with two or more 1's are illegal.

The assigned entries of identified GISes can indicate closely related FFs. Technique T6 exploits this fact. It uses the state generator (also used by GIS-learning) on the assigned entries of an identified GIS (which has no more than a predefined number of assigned entries) to generate PGISes to be justified by the STPG. For example, if state X1XX01 is detected to be a GIS, T6 generates the 7 remaining PGISes X0XX00, X0XX01, X0XX10, \ldots, X1XX11, which are tried to be justified. Another example: If during STPG a GIS is detected, which is one of the GISes of a BCD counter (e.g., 1011), than also the other GISes of the BCD counter are found using this technique.
7.6 Experimental results

The ATPG system DAT for combinational and sequential circuits, implemented in C++, has been extended with the techniques proposed in the previous sections. To evaluate the effectiveness and efficiency, it has been applied to the ISCAS'89 benchmark circuits [Brg89] and some sequential 3-state circuits. The experiments have been run on a Pentium Pro 200Mhz processor with 256Mb main-memory under a Linux operating system (Specint95: 8.09 [Spe]).

ATPG for sequential circuits has been performed according to the description in Section 5.7. The PSL signal model has been used. The time limit of the total ATPG job was 4 hours. Static learning to learn global implications (Section 5.4.1) was limited to 15 minutes. GIS-learning (Section 7.2) was performed with a time limit of 1 hour. If the GIS list was complete (all possible GISes are found), then during the remaining ATPG process no attempt was made anymore to find or optimize GISes. Between the first two STPG stages, learning to find unjustifiable signal lines (Section 5.4.2) was performed with a time limit of 15 minutes, and process EIS (using technique T1) was performed, limited to 15 minutes. For this process, the GIS list is sorted on the number of assignments (highest first). However, when technique T4 (direct expansion) is performed during STPG, the usefulness of this process is limited to those GISes which could not be expanded by T4 due to the much lower backtrack limit of T4 (i.e., these GISes are not marked prime; see Section 7.5, Page 154).

Seven experiments have been performed. In the first experiment (T.all), ATPG is performed using all techniques as described in Section 7.5, on all ISCAS'89 circuits and some sequential 3-state circuits. The results of this experiment are compared to the results of HITEC [Nie91], MOSAIC [Dar97], and DUST [Gou93]. To show the usefulness of the proposed techniques, the second to seventh experiments perform sequential ATPG as in experiment T.all, except that each time one or more of the techniques are deactivated:

- Experiment T.0: Sequential ATPG as experiment T.all. However, all techniques (T1 . . . T6) are deactivated. Furthermore, no GIS learning is performed; i.e., no effort is performed to explicitly find GISes and/or optimize the implicitly (during STPG) identified GISes.

- Experiment T.123: Sequential ATPG as experiment T.all. However, techniques T4, T5 and T6 are deactivated.

- Experiment T.3: Sequential ATPG as experiment T.all, however, without technique T3.

- Experiment T.4: Sequential ATPG as experiment T.all, however, without technique T4.

- Experiment T.5: Sequential ATPG as experiment T.all, however, without technique T5.

- Experiment T.6: Sequential ATPG as experiment T.all, however, without technique T6.
Chapter 7. Illegal state space identification

Experiments $T_{all}$, $T_{123}$ and $T_{0}$: Sequential circuit ATPG. Experiment $T_{all}$ is performed as described in previous paragraph. Technique T4 (direct expansion) is performed during GIS-learning when the number of assignments of a GIS is 10 or higher; and during the ATPG stages when the number of assignments of a GIS exceeds 50% of the FFs. These parameters have been experimentally determined and help to avoid that the ATPG system spends too much time in GIS expansion. For this reason also a low back-track limit (16) was chosen for technique T4. Technique T6 (enumerate assigned part of a GIS) is performed when a GIS has 4 or fewer assignments. This parameter has also been determined experimentally; a higher value results in too many PGISeS and hence too long computing times. Table 7.3 shows the experimental results for the ISCAS’89 circuits. These results are compared to the results of experiments $T_{123}$ and $T_{0}$ which are also shown in Table 7.3. Column CPU reports the total CPU time for creation and initialization of the circuit, for performing the processes before and during test generation, and for test generation and fault simulation. The bold values in the table indicate a significant improvement for a circuit, when comparing experiment $T_{all}$ to $T_{123}$, considering the CPU times (column CPU), reached fault efficiencies (column f.e.), and the number of GISes (column #GIS).

The bold values in the table show that experiment $T_{all}$, which uses all proposed techniques, often is significantly better than experiment $T_{123}$. For example, for circuits s382, s526, s1494 and s35932 the CPU times are reduced, and for the circuits s526, s1423, s5378, s38584.1 fault efficiencies are improved. The total CPU costs of experiment $T_{all}$ have been reduced with 26% compared to $T_{123}$ (see row 'SUM_A' in Table 7.3). In addition, the total fault efficiency reached by experiment $T_{all}$ have been increased with 4.7% compared to experiment $T_{123}$.

The fault efficiencies reached for the larger ISCAS circuits are often significantly improved (see for example circuits s15850.1, s38584, and s38584.1). However, they are still not high. The high fault efficiencies obtained for circuits c13207, s13207.1, s15850 are due to the high number of untestable faults: Many signal lines of these circuits are not initializable. Most of these uninitializable signal lines are identified during the pre-processes (see Section 5.1).

For some circuits the CPU times for exp. $T_{all}$ increased (e.g., s349, s386, and s838) compared to experiment $T_{123}$. This is due to the techniques T4, T5, T6, which are not (or less) useful for these circuits (Note: For some circuits also the techniques T1, T2, and T3 were not useful, because these circuits are easy to handle and do not require GIS identification and/or optimization. In that case the results of experiment $T_{0}$ are best (e.g., s510 and s9234)).

Table 7.3 shows that for only eight of the first 25 benchmark circuits, 100% fault efficiency is not yet reached. Usually when 100% fault efficiency has been reached for a circuit, the corresponding CPU costs have been low. Circuits with almost 100% fault efficiency often used much more CPU time (e.g., circuit s382, s400, and s444). Examination of the ATPG process for these circuits showed that often much CPU time is spent for a very small group of very hard faults (e.g., 1 fault in s382; 1 fault in s400; etc.), in the last, very "CPU expensive", ATPG stages. However, for circuits s382, s400, and s444, 100% fault efficiency can be reached with a higher back-track limit in the last ATPG stage; at the
expense of CPU time (see the footnote of Table 7.3).

Circuits s382, s400, s444, s526 and s526n have very long test sequences (see Table 7.3) because these circuits, which are traffic light controllers [Brg89], contain a timer, implemented with (BCD-)counters. Some faults require a certain state on these counters, which has to be reached starting from the reset-state of these counters, sometimes resulting in long test sequences.

Although circuits s820, s832, s1488 and s1494 have very few flipflops (5 or 6), and during GIS learning the complete GIS list has been determined, still it is very hard for DAT to reach a high fault efficiency. We noticed that often many states which have to be justified are fault dependent. We expect that applying IS optimization (IS merging, identification, and expansion) on fault-dependent ISes will improve the fault efficiencies for these circuits. Furthermore, we expect also that applying STPG within a window of timeframes (see Section 4.4, Page 67) will improve the STPG performance for these circuits, because HITEC, which applies this window-technique has very good results for these circuits.

The sizes of the GIS lists are often reduced (see bold entries in column \#GIS of exp. T.all), although they cover a larger GIS space, due to the sparser assigned GISes (more GISes are prime). The smaller GIS lists often improve the performance of the ATPG, because fewer GISes have to be checked against the current states (e.g., see results of s1423, s5378, and s35932). Due to the smaller GIS lists and usage of cvectors (see Section 7.4), also the memory usage often is low.

Comparing experiments T.all/T.123 with experiment T.0 shows that the CPU costs and/or fault efficiencies are much worse for experiment T.0. The total CPU costs of experiment T.all (T.123) have been reduced with 37% (16%) compared to experiment T.0 (see row 'SUM_A' in Table 7.3). In addition, the total fault efficiency reached by experiment T.all (T.123) have been increased with 5.7% (1%) compared to experiment T.0. The comparisons prove the effectiveness and efficiency of our approach to constrain the STPG by (improved) determination of the illegal search space (by GIS identification and expansion).

Table 7.4 shows the results of DAT (experiment T.all) in comparison with HITEC [Rud95] (Machine: HP9000 J200 with 256 MB memory; SpecInt95: 4.98), MOSAIC [Dar97] (Machine: Sun Sparc20-70MHz; SpecInt95: 3.11) and DUST [Gou93] (Machine: Sun 4/60; SpecInt95: 2.5). The table shows that sequential ATPG with DAT almost constantly results in better fault efficiencies, often combined with lower CPU costs (for example, circuits s382, s526, and s5378).

Rows 'SUM.H1', 'SUM.M1', and 'SUM.D1' show the total CPU costs and fault efficiency of DAT and those from HITEC, MOSAIC and DUST respectively. The total CPU costs of HITEC, MOSAIC and DUST are modified based on the SpecInt95 values of the applied machines, to allow direct comparison between the various sums. The total fault efficiency is the fraction of the summation of the number of detected and proven untestable faults, and the summation of the total number of faults of the considered circuits; i.e., larger circuits which have more faults have a larger impact on (or weight in) the total fault efficiency. Therefore, the largest four circuits are excluded from these sums (see
Chapter 7. Illegal state space identification

rows 'SUM_H2', 'SUM_M2', and 'SUM_D2'). The total fault efficiency reached by DAT is significantly higher than the total fault efficiencies reached by the other ATPG systems. In addition, the total CPU usage of DAT is lower compared to the CPU usage of HITEC. However, DAT's total CPU usage is higher compared to the CPU usage of MOSAIC and DUST. This larger CPU usage of DAT (compared to MOSAIC and DUST) usually is due to the much higher fault efficiencies reached by DAT; i.e., the ATPG systems MOSAIC and DUST often could not handle the hardest faults.

Table 7.5 shows the results of experiment T_all for the sequential 3-state circuits. The number in the circuit name expresses the number of signal lines in the circuit. Details on these circuits can be found in Appendix B. The fault efficiencies that are reached for these industrial circuits are very high and the CPU costs are low. The effectiveness and efficiency of DAT for these circuits is often better than for similar sized ISCAS'89 circuits. This may indicate that the complexity of the ISCAS'89 circuits is often much higher than the industrial circuits.

Experiments T_3, T_4, T_5, T_6: ATPG without T3, T4, T5 and T6, respectively. Table 7.6 shows the results for these experiments, which are equal to experiment T_all, except that technique T3, T4, T5, or T6 is deactivated. (A description for the columns in this table can be found below Table 7.3). The experiments are performed to show that the proposed techniques are effective. For each experiment, the table gives the results for a number of circuits for which the technique is remarkably effective or ineffective. For comparison, the table shows the results of experiment T_all (column 8 and 9). The last column in the table, if marked with a √, indicates that a proposed technique has been useful for that circuit. The results show that each technique is useful for a number of circuits; i.e., the effectiveness of the techniques is circuit dependent. Using this observation, the following can be concluded:

- Circuit analysis can help the ATPG system to determine which technique would be useful. In addition, the ATPG process could decide during this ATPG process which technique to use: E.g., if technique T3 (common fraction array) often does not succeed in identifying new GISes then do not use this technique anymore during the remaining ATPG process. Similar decisions can be made for the remaining techniques.

- It is very important that the parameters to steer the processes for GIS identification and expansion are chosen well, in order to avoid that too much time is spend in them. E.g., for technique T4 it is important that the value for the minimum number of assignments is well chosen. If the value is chosen too high, fewer GISes are expanded, resulting in a smaller detected GIS space. If the value is chosen too low, too many GISes are expanded, resulting in high CPU costs for GIS expansion. Both cases often result in a higher total CPU usage and reduced fault efficiencies.
7.7 Summary and conclusions

An ATPG system (DAT) for sequential circuits has been presented; it focuses on discovering the Global Illegal State (GIS) space, which constrains the search space during STPG to justifiable states. The GIS space is gradually discovered implicitly during STPG for faults (see Section 5.2.7.1). However, these "accidentally" found GISes are often over-specified; i.e., are not prime. Therefore, six techniques have been proposed to expand the known GIS space by trying to remove the over-specification. By using the current GIS list, they generate Potential GISes (PGIs) based on the concepts of GIS identification, merging and expansion. These PGIs are tried to be justified using the Justifier (the Justifier is phase 3 of the STPG (the backward justification phase)). A justification failure for a PGIS results in a new GIS, which usually covers one or more GISes from the original GIS list. This reduces the size of the GIS list (thus saves in memory usage) and expands the known GIS space (thus increases the performance of the STPG). The six techniques are:

\textit{T1:} Expansion of single GISes by unassigning single entries, used in process EIS (solving the problem of very large GIS lists covering a small part of the GIS space).

\textit{T2:} PGIS generation from two similar GISes by finding common assigned fragments, used in process CIS.

\textit{T3:} PGIS generation by determining the common assigned fragments of a range of GISes identified during STPG for one fault (solving the problem that STPG may find and store many similar GISes).

\textit{T4:} Direct expansion of new GISes; i.e., removal of over-specified entries of new GISes as soon as they are stored in the GIS list (solving the problem that GISes are stored, which would not be stored when already stored GISes had been minimized).

\textit{T5:} PGIS generation to preserve existing FF-groups.

\textit{T6:} PGIS generation by enumerating the assigned part of a GIS (to discover BCD counters, one-hot codes, etc.).

To improve the comparison speed of two states, FF-groups and cvectors have been proposed. Cvectors also reduce the storage requirements for the GISes up to a factor 10.

The process GIS-learning has been proposed to find sparsely assigned GISes (which are often prime). Experimental results show that GIS-learning often can identify a very large part (or all) of the GIS space, within low computing times. The low computing times are mainly due to the legal state cache, which has been proposed to speed-up the learning process and the optimization techniques.

The experimental results of test generation demonstrate that:

- 100% fault efficiency for a large number of the ISCAS’89 and 3-state industrial circuits has been reached within reasonable CPU time; for 20 out of the 25 first ISCAS’89 circuits 100% fault efficiency have been reached. For only 6 out of the 38 ISCAS’89 circuits the reached fault efficiencies are below 90%. 
• Better results in CPU usage and/or fault efficiency have been obtained, compared to other ATPG systems (HITEC, MOSAIC and DUST).

• The proposed techniques are very effective. However, the usefulness of a particular technique is highly circuit dependent. Therefore, it would be desirable to make the selection of the techniques circuit dependent.

• There is still a lot of room for further improvement in STPG. For some (ISCAS) circuits high fault efficiencies are still to be reached.

Finally, for future extensions we intend to develop further techniques for GIS identification and expansion, and techniques to select which GIS identification and expansion techniques are useful for a circuit. Furthermore, we expect that for very hard faults, the use of the IS identification and expansion techniques in the fault-dependent state space, will yield significant test generation improvements.
## 7.7. Summary and conclusions

<table>
<thead>
<tr>
<th>Circuit</th>
<th>#flts</th>
<th>#st</th>
<th>#unt</th>
<th>seql</th>
<th>#GIS</th>
<th>CPU</th>
<th>%fe</th>
<th>Exp. T.123</th>
<th>Exp. T.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(T1, T2, T3, T4, T5 and T6)</td>
<td>no Tx</td>
</tr>
<tr>
<td>s27</td>
<td>32</td>
<td>32</td>
<td>0</td>
<td>13</td>
<td>1</td>
<td>1.17</td>
<td>100</td>
<td>1</td>
<td>1.11</td>
</tr>
<tr>
<td>s208</td>
<td>215</td>
<td>137</td>
<td>78</td>
<td>148</td>
<td>7</td>
<td>1.67</td>
<td>100</td>
<td>7</td>
<td>6.68</td>
</tr>
<tr>
<td>s208.1</td>
<td>217</td>
<td>18</td>
<td>199</td>
<td>11</td>
<td>8</td>
<td>2.11</td>
<td>100</td>
<td>8</td>
<td>1.79</td>
</tr>
<tr>
<td>s298</td>
<td>308</td>
<td>265</td>
<td>43</td>
<td>173</td>
<td>25</td>
<td>5.16</td>
<td>100</td>
<td>29</td>
<td>18.30</td>
</tr>
<tr>
<td>s344</td>
<td>342</td>
<td>329</td>
<td>13</td>
<td>101</td>
<td>12</td>
<td>56.82</td>
<td>100</td>
<td>484</td>
<td>89.42</td>
</tr>
<tr>
<td>s349</td>
<td>350</td>
<td>335</td>
<td>15</td>
<td>139</td>
<td>13</td>
<td>84.95</td>
<td>100</td>
<td>134</td>
<td>70.26</td>
</tr>
<tr>
<td>s382</td>
<td>399</td>
<td>364</td>
<td>34</td>
<td>209</td>
<td>161</td>
<td>1655*</td>
<td>99.75</td>
<td>27</td>
<td>5327</td>
</tr>
<tr>
<td>s386</td>
<td>384</td>
<td>314</td>
<td>70</td>
<td>294</td>
<td>20</td>
<td>307.4</td>
<td>100</td>
<td>10</td>
<td>263.7</td>
</tr>
<tr>
<td>s400</td>
<td>424</td>
<td>382</td>
<td>41</td>
<td>2008</td>
<td>161</td>
<td>1683*</td>
<td>99.76</td>
<td>27</td>
<td>4743</td>
</tr>
<tr>
<td>s420</td>
<td>430</td>
<td>179</td>
<td>251</td>
<td>170</td>
<td>15</td>
<td>3.26</td>
<td>100</td>
<td>15</td>
<td>25.18</td>
</tr>
<tr>
<td>s420.1</td>
<td>455</td>
<td>28</td>
<td>427</td>
<td>15</td>
<td>16</td>
<td>8.46</td>
<td>100</td>
<td>16</td>
<td>4.38</td>
</tr>
<tr>
<td>s444</td>
<td>474</td>
<td>424</td>
<td>49</td>
<td>1712</td>
<td>161</td>
<td>2060*</td>
<td>99.79</td>
<td>27</td>
<td>4372</td>
</tr>
<tr>
<td>s510</td>
<td>564</td>
<td>0</td>
<td>564</td>
<td>0</td>
<td>0</td>
<td>3.27</td>
<td>100</td>
<td>0</td>
<td>1.65</td>
</tr>
<tr>
<td>s526</td>
<td>555</td>
<td>451</td>
<td>101</td>
<td>6842</td>
<td>189</td>
<td>3513</td>
<td>99.46</td>
<td>27</td>
<td>9019</td>
</tr>
<tr>
<td>s526n</td>
<td>553</td>
<td>452</td>
<td>99</td>
<td>6071</td>
<td>191</td>
<td>3205</td>
<td>99.64</td>
<td>27</td>
<td>9019</td>
</tr>
<tr>
<td>s641</td>
<td>467</td>
<td>404</td>
<td>63</td>
<td>170</td>
<td>22</td>
<td>35.68</td>
<td>100</td>
<td>26</td>
<td>931.6</td>
</tr>
<tr>
<td>s713</td>
<td>581</td>
<td>476</td>
<td>105</td>
<td>112</td>
<td>42</td>
<td>175.1</td>
<td>100</td>
<td>26</td>
<td>879.3</td>
</tr>
<tr>
<td>s820</td>
<td>850</td>
<td>761</td>
<td>16</td>
<td>657</td>
<td>6</td>
<td>14401</td>
<td>91.41</td>
<td>6</td>
<td>14403</td>
</tr>
<tr>
<td>s832</td>
<td>870</td>
<td>767</td>
<td>33</td>
<td>616</td>
<td>6</td>
<td>14402</td>
<td>91.95</td>
<td>6</td>
<td>14402</td>
</tr>
<tr>
<td>s838</td>
<td>857</td>
<td>254</td>
<td>603</td>
<td>128</td>
<td>31</td>
<td>209.1</td>
<td>100</td>
<td>31</td>
<td>101.8</td>
</tr>
<tr>
<td>s838.1</td>
<td>931</td>
<td>48</td>
<td>883</td>
<td>12</td>
<td>32</td>
<td>83.14</td>
<td>100</td>
<td>32</td>
<td>20.39</td>
</tr>
<tr>
<td>s953</td>
<td>1079</td>
<td>90</td>
<td>889</td>
<td>26</td>
<td>0</td>
<td>5.12</td>
<td>100</td>
<td>0</td>
<td>4.91</td>
</tr>
<tr>
<td>s1196</td>
<td>1242</td>
<td>1239</td>
<td>3</td>
<td>354</td>
<td>11</td>
<td>6.59</td>
<td>100</td>
<td>11</td>
<td>15.60</td>
</tr>
<tr>
<td>s1238</td>
<td>1355</td>
<td>1283</td>
<td>72</td>
<td>359</td>
<td>11</td>
<td>7.51</td>
<td>100</td>
<td>11</td>
<td>25.10</td>
</tr>
<tr>
<td>s1423</td>
<td>1515</td>
<td>597</td>
<td>19</td>
<td>103</td>
<td>110</td>
<td>14401</td>
<td>40.66</td>
<td>17730</td>
<td>14403</td>
</tr>
<tr>
<td>s1488</td>
<td>1486</td>
<td>1426</td>
<td>26</td>
<td>1623</td>
<td>5</td>
<td>4776</td>
<td>97.71</td>
<td>5</td>
<td>14402</td>
</tr>
<tr>
<td>s1494</td>
<td>1506</td>
<td>1441</td>
<td>38</td>
<td>1947</td>
<td>5</td>
<td>5310</td>
<td>98.21</td>
<td>8</td>
<td>14403</td>
</tr>
<tr>
<td>s5378</td>
<td>4603</td>
<td>3555</td>
<td>660</td>
<td>730</td>
<td>185</td>
<td>14401</td>
<td>91.57</td>
<td>11912</td>
<td>14401</td>
</tr>
<tr>
<td>s9234</td>
<td>6927</td>
<td>18</td>
<td>6909</td>
<td>3</td>
<td>10</td>
<td>329.7</td>
<td>100</td>
<td>0</td>
<td>86.23</td>
</tr>
<tr>
<td>s9234.1</td>
<td>6927</td>
<td>470</td>
<td>3194</td>
<td>40</td>
<td>321</td>
<td>14402</td>
<td>52.89</td>
<td>17654</td>
<td>14403</td>
</tr>
<tr>
<td>s13207</td>
<td>9815</td>
<td>641</td>
<td>8738</td>
<td>319</td>
<td>3328</td>
<td>14404</td>
<td>95.56</td>
<td>41</td>
<td>14403</td>
</tr>
<tr>
<td>s13207.1</td>
<td>9815</td>
<td>602</td>
<td>9213</td>
<td>25</td>
<td>330</td>
<td>1555</td>
<td>100</td>
<td>3</td>
<td>4669</td>
</tr>
<tr>
<td>s15850</td>
<td>11725</td>
<td>85</td>
<td>11640</td>
<td>11</td>
<td>24</td>
<td>4603</td>
<td>100</td>
<td>8753</td>
<td>889</td>
</tr>
<tr>
<td>s15850.1</td>
<td>11726</td>
<td>4323</td>
<td>1373</td>
<td>1270</td>
<td>1053</td>
<td>14401</td>
<td>48.58</td>
<td>1059</td>
<td>14402</td>
</tr>
<tr>
<td>s35932</td>
<td>39094</td>
<td>35096</td>
<td>3984</td>
<td>247</td>
<td>95</td>
<td>6148</td>
<td>99.96</td>
<td>1472</td>
<td>14401</td>
</tr>
<tr>
<td>SUM.A</td>
<td>119072</td>
<td>57286</td>
<td>50545</td>
<td>28450</td>
<td>6607</td>
<td>136735</td>
<td>90.6</td>
<td>59625</td>
<td>184608</td>
</tr>
<tr>
<td>s38417</td>
<td>31180</td>
<td>13169</td>
<td>900</td>
<td>823</td>
<td>7</td>
<td>14402</td>
<td>7.11</td>
<td>3039</td>
<td>14402</td>
</tr>
<tr>
<td>s38584</td>
<td>36303</td>
<td>6614</td>
<td>6155</td>
<td>333</td>
<td>654</td>
<td>14401</td>
<td>35.17</td>
<td>10082</td>
<td>14472</td>
</tr>
<tr>
<td>s38584.1</td>
<td>36303</td>
<td>16426</td>
<td>1666</td>
<td>759</td>
<td>293</td>
<td>14401</td>
<td>49.84</td>
<td>15200</td>
<td>14441</td>
</tr>
</tbody>
</table>

#flts: total no. of faults  
#st: no. of faults detected  
#unt: no. of uncontestable faults  
seql: test sequence length  
CPU(s): CPU usage in seconds  
f.e. (%): total fault-efficiency  
#GIS: number of GISes after ATPG has been finished

* For circuits s382, s400, and s444 100% fault efficiency has been reached after 7532, 7157, 9252 CPU seconds respectively

## Table 7.3: Results of ATPG for ISCAS'89 circuits (Experiment T.all)
<table>
<thead>
<tr>
<th>Circuit</th>
<th>DAT</th>
<th>HITEC</th>
<th>MOSAIC</th>
<th>DUST</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU(s)</td>
<td>f.e. (%)</td>
<td>CPU(s)</td>
<td>f.e. (%)</td>
</tr>
<tr>
<td>s27</td>
<td>1</td>
<td>100</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>s208</td>
<td>2</td>
<td>100</td>
<td>100</td>
<td>94.5</td>
</tr>
<tr>
<td>s344</td>
<td>57</td>
<td>100</td>
<td>484</td>
<td>97.9</td>
</tr>
<tr>
<td>s349</td>
<td>85</td>
<td>100</td>
<td>464</td>
<td>98.6</td>
</tr>
<tr>
<td>s382</td>
<td>1655</td>
<td>99.8</td>
<td>5472</td>
<td>77.7</td>
</tr>
<tr>
<td>s386</td>
<td>307</td>
<td>100</td>
<td>7</td>
<td>100</td>
</tr>
<tr>
<td>s400</td>
<td>1683</td>
<td>99.8</td>
<td>4356</td>
<td>84.4</td>
</tr>
<tr>
<td>s420</td>
<td>3</td>
<td>100</td>
<td>552</td>
<td>90.9</td>
</tr>
<tr>
<td>s444</td>
<td>2060</td>
<td>99.8</td>
<td>5364</td>
<td>84.0</td>
</tr>
<tr>
<td>s510</td>
<td>3</td>
<td>100</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>s526</td>
<td>3513</td>
<td>99.5</td>
<td>20844</td>
<td>61.1</td>
</tr>
<tr>
<td>s526n</td>
<td>3205</td>
<td>99.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s641</td>
<td>36</td>
<td>100</td>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>s713</td>
<td>175</td>
<td>100</td>
<td>7</td>
<td>100</td>
</tr>
<tr>
<td>s820</td>
<td>14401</td>
<td>91.4</td>
<td>210</td>
<td>100</td>
</tr>
<tr>
<td>s832</td>
<td>14402</td>
<td>92.0</td>
<td>345</td>
<td>100</td>
</tr>
<tr>
<td>s838</td>
<td>209</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s953</td>
<td>13</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s1196</td>
<td>7</td>
<td>100</td>
<td>6</td>
<td>100</td>
</tr>
<tr>
<td>s1238</td>
<td>8</td>
<td>100</td>
<td>8</td>
<td>100</td>
</tr>
<tr>
<td>s1423</td>
<td>14401</td>
<td>40.7</td>
<td>50040</td>
<td>48.6</td>
</tr>
<tr>
<td>s1488</td>
<td>4776</td>
<td>97.7</td>
<td>990</td>
<td>99.9</td>
</tr>
<tr>
<td>s1494</td>
<td>5310</td>
<td>98.2</td>
<td>575</td>
<td>99.9</td>
</tr>
<tr>
<td>s5378</td>
<td>14401</td>
<td>91.6</td>
<td>66240</td>
<td>74.9</td>
</tr>
<tr>
<td>SUM_H1</td>
<td>77282</td>
<td>91.5</td>
<td>96269*</td>
<td>86.3</td>
</tr>
<tr>
<td>SUM_M1</td>
<td>80714</td>
<td>92.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUM_D1</td>
<td>66317</td>
<td>96.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s9234</td>
<td>330</td>
<td>100</td>
<td></td>
<td>0.1</td>
</tr>
<tr>
<td>s13207</td>
<td>14404</td>
<td>95.6</td>
<td></td>
<td>4680</td>
</tr>
<tr>
<td>s15850</td>
<td>889</td>
<td>100</td>
<td></td>
<td>246</td>
</tr>
<tr>
<td>s35932</td>
<td>6148</td>
<td>99.9</td>
<td>17028</td>
<td>99.4</td>
</tr>
<tr>
<td>SUM_H2</td>
<td>83430</td>
<td>97.3</td>
<td>106751*</td>
<td>95.3</td>
</tr>
<tr>
<td>SUM_M2</td>
<td>102485</td>
<td>97.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUM_D2</td>
<td>81940</td>
<td>97.7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CPU(s): CPU usage in seconds  
f.e.(%): Total fault efficiency  
*: Modified CPU costs based on the SpecInt95 values, to allow direct comparison between the various sums.

Table 7.4: Comparison of results of DAT, HITEC, DUST and MOSAIC
7.7. Summary and conclusions

<table>
<thead>
<tr>
<th>Circuit</th>
<th>#nflts</th>
<th>#tst</th>
<th>#unt</th>
<th>#GIS</th>
<th>seql</th>
<th>CPU(s)</th>
<th>f.e. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>p169</td>
<td>251</td>
<td>185</td>
<td>14</td>
<td>27</td>
<td>62</td>
<td>2.67</td>
<td>99.2</td>
</tr>
<tr>
<td>p314</td>
<td>312</td>
<td>300</td>
<td>12</td>
<td>2</td>
<td>198</td>
<td>98.9</td>
<td>100</td>
</tr>
<tr>
<td>p660</td>
<td>818</td>
<td>765</td>
<td>37</td>
<td>0</td>
<td>187</td>
<td>12.1</td>
<td>98.3</td>
</tr>
<tr>
<td>p1514</td>
<td>1708</td>
<td>1585</td>
<td>31</td>
<td>71</td>
<td>174</td>
<td>14403</td>
<td>94.3</td>
</tr>
<tr>
<td>p2024</td>
<td>2140</td>
<td>1945</td>
<td>194</td>
<td>48</td>
<td>567</td>
<td>187</td>
<td>99.9</td>
</tr>
<tr>
<td>p2133</td>
<td>2375</td>
<td>2193</td>
<td>182</td>
<td>7</td>
<td>117</td>
<td>4.42</td>
<td>100</td>
</tr>
<tr>
<td>p2756</td>
<td>2956</td>
<td>2540</td>
<td>416</td>
<td>8</td>
<td>604</td>
<td>4380</td>
<td>100</td>
</tr>
<tr>
<td>p7273</td>
<td>7603</td>
<td>761</td>
<td>6842</td>
<td>3</td>
<td>3407</td>
<td>357</td>
<td>100</td>
</tr>
<tr>
<td>p24938</td>
<td>22439</td>
<td>39</td>
<td>22403</td>
<td>12</td>
<td>15</td>
<td>1046</td>
<td>100</td>
</tr>
<tr>
<td>SUM</td>
<td>40602</td>
<td>10313</td>
<td>30131</td>
<td>178</td>
<td>5331</td>
<td>20491</td>
<td>99.6</td>
</tr>
</tbody>
</table>

#nflts: Total number of faults  #tst: Number of faults detected
#unt: Number of untestable faults  seql: Test sequence length
CPU(s): CPU usage in seconds  f.e. (%): Total fault efficiency
#GIS: Number of GISes after ATPG has been finished

Table 7.5: Results of ATPG for sequential 3-state circuits

<table>
<thead>
<tr>
<th>Circuit</th>
<th>#tst</th>
<th>#unt</th>
<th>#gis</th>
<th>seql</th>
<th>CPU(s)</th>
<th>f.e. (%)</th>
<th>CPU(s)</th>
<th>f.e. (%)</th>
<th>useful</th>
</tr>
</thead>
<tbody>
<tr>
<td>s298</td>
<td>265</td>
<td>43</td>
<td>27</td>
<td>166</td>
<td>5.1</td>
<td>100</td>
<td>5.16</td>
<td>100</td>
<td>–</td>
</tr>
<tr>
<td>s349</td>
<td>335</td>
<td>15</td>
<td>54</td>
<td>122</td>
<td>242.1</td>
<td>100</td>
<td>84.95</td>
<td>100</td>
<td>✓</td>
</tr>
<tr>
<td>s382</td>
<td>364</td>
<td>34</td>
<td>27</td>
<td>3527</td>
<td>3919</td>
<td>99.8</td>
<td>1655</td>
<td>99.9</td>
<td>✓</td>
</tr>
<tr>
<td>s526</td>
<td>451</td>
<td>101</td>
<td>38</td>
<td>10735</td>
<td>8538</td>
<td>99.5</td>
<td>3513</td>
<td>99.5</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Circuit</th>
<th>#tst</th>
<th>#unt</th>
<th>#gis</th>
<th>seql</th>
<th>CPU(s)</th>
<th>f.e. (%)</th>
<th>CPU(s)</th>
<th>f.e. (%)</th>
<th>useful</th>
</tr>
</thead>
<tbody>
<tr>
<td>s349</td>
<td>335</td>
<td>15</td>
<td>1</td>
<td>122</td>
<td>236.8</td>
<td>100</td>
<td>56.8</td>
<td>100</td>
<td>✓</td>
</tr>
<tr>
<td>s641</td>
<td>404</td>
<td>63</td>
<td>7</td>
<td>249</td>
<td>8554</td>
<td>100</td>
<td>35.7</td>
<td>100</td>
<td>✓</td>
</tr>
<tr>
<td>s713</td>
<td>476</td>
<td>105</td>
<td>8</td>
<td>213</td>
<td>125.4</td>
<td>100</td>
<td>175.1</td>
<td>100</td>
<td>–</td>
</tr>
<tr>
<td>s5378</td>
<td>2740</td>
<td>605</td>
<td>7550</td>
<td>103</td>
<td>14401</td>
<td>72.7</td>
<td>14401</td>
<td>91.6</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Circuit</th>
<th>#tst</th>
<th>#unt</th>
<th>#gis</th>
<th>seql</th>
<th>CPU(s)</th>
<th>f.e. (%)</th>
<th>CPU(s)</th>
<th>f.e. (%)</th>
<th>useful</th>
</tr>
</thead>
<tbody>
<tr>
<td>s382</td>
<td>364</td>
<td>34</td>
<td>1</td>
<td>3527</td>
<td>4355</td>
<td>99.8</td>
<td>1655</td>
<td>99.8</td>
<td>✓</td>
</tr>
<tr>
<td>s526</td>
<td>451</td>
<td>101</td>
<td>7</td>
<td>10735</td>
<td>8627</td>
<td>99.5</td>
<td>3513</td>
<td>99.5</td>
<td>✓</td>
</tr>
<tr>
<td>s641</td>
<td>404</td>
<td>63</td>
<td>8</td>
<td>249</td>
<td>8517</td>
<td>100</td>
<td>35.7</td>
<td>100</td>
<td>✓</td>
</tr>
<tr>
<td>s5378</td>
<td>3516</td>
<td>645</td>
<td>69</td>
<td>634</td>
<td>14401</td>
<td>90.4</td>
<td>14401</td>
<td>91.6</td>
<td>–</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Circuit</th>
<th>#tst</th>
<th>#unt</th>
<th>#gis</th>
<th>seql</th>
<th>CPU(s)</th>
<th>f.e. (%)</th>
<th>CPU(s)</th>
<th>f.e. (%)</th>
<th>useful</th>
</tr>
</thead>
<tbody>
<tr>
<td>s349</td>
<td>335</td>
<td>15</td>
<td>42</td>
<td>122</td>
<td>250.9</td>
<td>100</td>
<td>84.95</td>
<td>100</td>
<td>✓</td>
</tr>
<tr>
<td>s444</td>
<td>424</td>
<td>49</td>
<td>27</td>
<td>3254</td>
<td>1550</td>
<td>99.8</td>
<td>2060</td>
<td>99.8</td>
<td>–</td>
</tr>
<tr>
<td>s526</td>
<td>451</td>
<td>101</td>
<td>36</td>
<td>10735</td>
<td>8065</td>
<td>99.5</td>
<td>3513</td>
<td>99.5</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 7.6: Results of ATPG with deactivated T3, T4, T5 or T6
Chapter 8

Case study: ATPG for the Philips 80C51

In this chapter a case study is presented to demonstrate the effectiveness and efficiency of the ATPG system, described in the previous chapters. The considered case is the Philips 80C51 micro-controller [Phil97] for which ATPG has been performed on a (near) full-scan version, on a fully sequential version (non-scan), and on several partial-scan versions. The fault coverage of the full-scan version is above 90%, while the fault coverage of the non-scan version is almost zero. Therefore, scan FF selection has been performed to increase the fault coverage to the fault coverage level of the full-scan version. However, an as small as possible set of scan FFs has to be selected to save on chip area and test application time. FFs have been selected based on uninitializable, unobservable, hard-to-control, and/or hard-to-observe signal lines, based on aborted faults, and based on STPG results of modules within the 80C51. Experimental results, using partial-scan versions of the 80C51, demonstrate that almost 50% of the FFs has to be scannable in order to approach the fault coverage for the full-scan version. The fault efficiency reduces with ±10% percent, when ±30% FFs have been selected for scan.

This chapter is organized as follows: Section 8.1 introduces the case, assumptions, the target, and how the case is going to be handled. Section 8.2 describes the 80C51 and its modules. Section 8.3 presents the results for the (near) full-scan versions of the 80C51 and a number of the modules. Section 8.4 presents the results for the non-scan versions. Section 8.5 describes the scan selection methods and presents their effect on STPG for the 80C51, and a number of modules. Section 8.6 summarizes the results, concludes this case study, and presents recommendations for future research.
Figure 8.1: 80C51 micro-controller family is widely used in many applications

8.1 Introduction

This section introduces ATPG for the 80C51 micro-controller. The motivation, the target of this case study, and assumptions are described.

This case study has been performed on the DAT ATPG system for sequential circuits, which has been described in the previous chapters. Over the last three years the ATPG system DAT has been extended to handle sequential (Boolean and 3-state) circuits. Much effort has been put in achieving high fault efficiencies by generating test sets for the sequential ISCAS’89 [Brg89] circuits. The experimental results in Chapters 6 and 7 demonstrate the efficiency of this system. However, the ISCAS’89 benchmark circuits are only based on real industrial circuits; they have been modified by making them full Boolean and (partly) unrecognizable. In addition, most of the benchmark circuits are small, compared to real industrial circuits. Therefore, the motivation for this case study: ATPG on the sequential 80C51 circuit, which is a real industrial circuit.

The 80C51 of Philips [Phl97] comprises a family of 8-bit (and 16-bit) micro-controllers, which are applied in a wide range of electronic applications, such as cordless and portable telephones, remote controls for televisions, video-recorders, automobiles, lighting systems, medical systems, etc. (see Figure 8.1); anywhere a low-cost control system is needed, the 80C51 can be applied. It is shipped as a “stand-alone” IC, and also as an embedded core inside more complex ICs, and is based on CMOS as well as NMOS technologies. The 80C51 family consists of many types, versions, and variants. Each of them is dedicated to certain applications, and/or has been optimized for certain performance characteristics (e.g., low power, low voltage, high speed), and/or has unique properties/features (e.g., memory size and type: RAM, ROM, EPROM; special interfaces) or functions. E.g., 80C51 circuits applied in portable telephones are optimized for power consumption; these circuits have functions to deactivate parts of the circuit when the telephone is not used. One member of the 80C51 family is subject of our case study (see Section 8.2 for a detailed description).

The main target of this study is to evaluate the efficiency and effectiveness, but also to identify shortcomings and problems, of the DAT ATPG system for sequential circuits, using the 80C51 circuit core. In addition, inside several Philips departments, there is a
8.2. Circuit description

In this section, the specific 80C51 circuit that is the subject of this case study, is described. The essential building block of the 80C51 is the core (see Figure 8.2), which consists of: The 8-bit CPU, 32 bidirectional I/O lines in four ports, two 16-bit timers/counters, 128 bytes of on-board data RAM, 4K bytes of on-chip ROM program memory, and a

---

1A circuit model with an explicit scan-chain has explicit PIs for scan-input and scan-enable, and an explicit PO for scan-output. In addition, logic (such as multiplexers) and wiring is explicitly incorporated in the circuit model for the scan-chain function of the FFs. At this moment, DAT is capable to handle circuits with explicit scan-chains (it can generate explicit chain(s) if required). However, the heuristics to steer the decision-making process during STPG, to make a choice between using the scan-chain or not, have not been completed yet. Explicit scan-chains open the possibility to perform automatic scan-selection during STPG; i.e., the scan-chain is only used when required during test. In addition, the main-clock does not have to be stopped when scan-shifts are performed; i.e., it is allowed that the state of the non-scan FFs changes during scan-shifts.

2The 80C51 has 7 FFs with asynchronous (re)set, 6 of them are also allowed to be scannable. Ignoring the (re)set may worsen the fault coverage, however, the results of STPG with the (near) full-scan version demonstrate that the impact of ignoring the (re)set pins is low (fault coverage is > 90%).
2-priority interrupt controller. In addition, each 80C51 has application specific functions. The version used for this study has a UART serial interface, an I2C interface, and features to handle power consumption (WDT (Watch Dog Timer), PCON, WDTKEY), as shown in Figure 8.2.

Table 8.1 shows some characteristic numbers for the 80C51 (used in this study). The 80C51 contains 24610 signal lines, 11756 Boolean gates and 105 3-state elements. Of the PIs, 23 out of the 25 PIs are fixed to U; i.e., they are not allowed to be controlled during STPG, while these PIs can only be controlled by the ROM and HOST\(^3\); the ROM has been removed from the circuit because the STPG cannot handle memory. The RAM has been remodeled as a buffer; i.e., all data on the inputs of the RAM is “immediately” visible on the corresponding outputs of the RAM. The circuit has one PO, and 35 I/O terminals (the 80C51 has four 8-bit I/O ports and 3 bi-directional control terminals).

The circuit has 56 logic levels, which makes it a relatively deep circuit. A large fraction of the signal lines are allowed to become U, hence can be influenced by (non-scan) FFs (24469 signal lines). The FFRs are (as usual) very small: On average they consists of 3.13 elements and have 3.37 inputs. The number of inputs of the elements range from one up to eight; on average an element has 1.75 inputs, which is low (4300 elements have only one input).

The number of FFs is 807 of which 806 FFs are allowed to become scannable. The remaining FF is part of the clock-driver (which also controls the scan-chain) and therefore cannot be scannable. Hence, even for the near full-scan version of the 80C51, a TPG for

\(^3\)i.e., an application, which communicates with, and makes use of the 80C51; such as a DSP.
8.3 Test generation in full-scan mode

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
<th>#conn</th>
<th>#FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORE</td>
<td>Core of the 80C51</td>
<td>16430</td>
<td>475</td>
</tr>
<tr>
<td>-CPU</td>
<td>Central processing unit</td>
<td>7904</td>
<td>184</td>
</tr>
<tr>
<td>-ALU</td>
<td>Arithmetic logic unit</td>
<td>2331</td>
<td>39</td>
</tr>
<tr>
<td>-INT7L2</td>
<td>7 vector/2 priority level interrupt controller</td>
<td>3395</td>
<td>91</td>
</tr>
<tr>
<td>ASFs</td>
<td>Application specific functions</td>
<td>8412</td>
<td>314</td>
</tr>
<tr>
<td>-UART</td>
<td>Serial interface</td>
<td>1324</td>
<td>65</td>
</tr>
<tr>
<td>-I2C</td>
<td>Bus master/slave interface</td>
<td>1993</td>
<td>65</td>
</tr>
<tr>
<td>-WDT</td>
<td>Watch dog timer</td>
<td>583</td>
<td>29</td>
</tr>
<tr>
<td>-TIMER2</td>
<td>16-bit timer/counter</td>
<td>2235</td>
<td>52</td>
</tr>
</tbody>
</table>

Table 8.2: Characteristic numbers of the 80C51 modules

A sequential circuit is required to avoid that faults are incorrectly marked untestable. In the full-scan version, the 80C51 has eight scan-chains; the scan-chain length ranges from 16 FFs up to 184 FFs. However, during STPG we do not consider the scan-chains explicitly (see Section 8.1), so the scan-chain lengths are not relevant during STPG.

The circuit can operate in normal-mode or test-mode. In normal-mode four clocks control the FFs. In test-mode the number of clocks has been reduced to one⁴, and the scan-chains are available; test generation has been performed in test-mode.

Table 8.2 lists the main modules of the 80C51 (see also Figure 8.2), together with a short description, the number of signal lines (\#conn), and the number of FFs. For these modules, CTPG has been performed with the full-scan versions (i.e., all FFs are selected as SFF), and STPG with the non-scan and the partial-scan versions.

8.3 Test generation in full-scan mode

This section presents the experimental results of STPG for the full-scan versions of the 80C51 (80C51F) and the modules in the 80C51 as has been listed in Table 8.2. Sequential circuit ATPG has been performed on the 80C51F, as described in Chapters 5, 6, and 7, because this circuit has one D-FF which is not allowed to be scanable (near full-scan).

Combinational circuit ATPG has been performed on the modules of the 80C51; they are full-scan circuits. Table 8.3 shows the results. Very high fault-efficiencies and fault coverages for all circuits have been reached. The fault coverages shown in the table form the upper-bound for the fault coverages reached by STPG on partial-scan versions. Hence, for the 80C51, the fault coverage of 91.8% is the upper-bound.

Note that for the 80C51F more than 8% of the faults are untestable. This is mainly due to the fixed vector on the PIs, resulting in that many signal lines cannot be controlled and hence the corresponding fault(s) are untestable. The total computing time of the 80C51F has been more than 10 hours; however, a large amount of this time has been spent on the few very hard faults; a fault efficiency of 98% has been reached within 30 minutes.

⁴However, we assume a main-clock to control the FFs, and a test-clock to control the scan-chains; see Section 8.1.
### Table 8.3: Experimental results 80C51 circuit and 80C51 modules (full-scan)

<table>
<thead>
<tr>
<th>Name</th>
<th>#flt</th>
<th>#tst</th>
<th>#unt</th>
<th>#apt</th>
<th>#vec</th>
<th>fcover (%)</th>
<th>feff (%)</th>
<th>CPU (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORE</td>
<td>16130</td>
<td>15835</td>
<td>295</td>
<td>0</td>
<td>968</td>
<td>98.17</td>
<td>100.00</td>
<td>1319.99</td>
</tr>
<tr>
<td>CPU</td>
<td>7712</td>
<td>7664</td>
<td>43</td>
<td>5</td>
<td>560</td>
<td>99.38</td>
<td>99.94</td>
<td>1350.13</td>
</tr>
<tr>
<td>ALU</td>
<td>2251</td>
<td>2226</td>
<td>23</td>
<td>2</td>
<td>135</td>
<td>98.89</td>
<td>99.91</td>
<td>44.12</td>
</tr>
<tr>
<td>INT7L2</td>
<td>3395</td>
<td>3384</td>
<td>11</td>
<td>0</td>
<td>186</td>
<td>99.68</td>
<td>100.00</td>
<td>75.64</td>
</tr>
<tr>
<td>ASF</td>
<td>8308</td>
<td>8290</td>
<td>18</td>
<td>0</td>
<td>485</td>
<td>99.78</td>
<td>100.00</td>
<td>23.97</td>
</tr>
<tr>
<td>UART</td>
<td>1349</td>
<td>1349</td>
<td>0</td>
<td>0</td>
<td>69</td>
<td>100.00</td>
<td>100.00</td>
<td>1.72</td>
</tr>
<tr>
<td>I2C</td>
<td>2022</td>
<td>2022</td>
<td>0</td>
<td>0</td>
<td>177</td>
<td>100.00</td>
<td>100.00</td>
<td>3.36</td>
</tr>
<tr>
<td>WDT</td>
<td>587</td>
<td>587</td>
<td>0</td>
<td>0</td>
<td>39</td>
<td>100.00</td>
<td>100.00</td>
<td>1.26</td>
</tr>
<tr>
<td>TIMER2</td>
<td>2130</td>
<td>2129</td>
<td>1</td>
<td>0</td>
<td>176</td>
<td>99.95</td>
<td>100.00</td>
<td>7.38</td>
</tr>
<tr>
<td>80C51F</td>
<td>24204</td>
<td>22206</td>
<td>1982</td>
<td>16</td>
<td>1564</td>
<td>91.8</td>
<td>99.9</td>
<td>38115</td>
</tr>
</tbody>
</table>

- #flt: Total number of faults
- #tst: Number of faults detected
- #unt: Number of untestable faults
- #apt: Number of aborted faults
- #vec: Test sequence length
- fcover: Fault coverage
- feff: Fault efficiency
- CPU: Total CPU costs

### Table 8.4: Experimental results 80C51 circuit and 80C51 modules (non-scan)

<table>
<thead>
<tr>
<th>Name</th>
<th>#flt</th>
<th>#tst</th>
<th>#unt</th>
<th>#apt</th>
<th>#vec</th>
<th>fcover (%)</th>
<th>feff (%)</th>
<th>CPU (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORE</td>
<td>16130</td>
<td>2242</td>
<td>480</td>
<td>13408</td>
<td>21</td>
<td>13.90</td>
<td>16.88</td>
<td>36007</td>
</tr>
<tr>
<td>CPU</td>
<td>7712</td>
<td>1694</td>
<td>58</td>
<td>5960</td>
<td>42</td>
<td>21.97</td>
<td>22.72</td>
<td>36004</td>
</tr>
<tr>
<td>ALU</td>
<td>2251</td>
<td>2223</td>
<td>27</td>
<td>1</td>
<td>509</td>
<td>98.76</td>
<td>99.96</td>
<td>13203</td>
</tr>
<tr>
<td>INT7L2</td>
<td>3395</td>
<td>3327</td>
<td>42</td>
<td>26</td>
<td>1983</td>
<td>98.00</td>
<td>99.23</td>
<td>36001</td>
</tr>
<tr>
<td>ASF</td>
<td>8308</td>
<td>5220</td>
<td>29</td>
<td>1</td>
<td>1063</td>
<td>62.83</td>
<td>63.18</td>
<td>36002</td>
</tr>
<tr>
<td>UART</td>
<td>1349</td>
<td>1109</td>
<td>23</td>
<td>217</td>
<td>1614</td>
<td>82.21</td>
<td>83.91</td>
<td>36001</td>
</tr>
<tr>
<td>I2C</td>
<td>2022</td>
<td>1027</td>
<td>17</td>
<td>978</td>
<td>194</td>
<td>50.79</td>
<td>51.63</td>
<td>36002</td>
</tr>
<tr>
<td>WDT</td>
<td>587</td>
<td>224</td>
<td>17</td>
<td>346</td>
<td>58</td>
<td>38.16</td>
<td>41.06</td>
<td>36001</td>
</tr>
<tr>
<td>WDT(2)</td>
<td>587</td>
<td>520</td>
<td>25</td>
<td>42</td>
<td>50537</td>
<td>88.58</td>
<td>92.80</td>
<td>72001</td>
</tr>
<tr>
<td>TIMER2</td>
<td>2130</td>
<td>2115</td>
<td>10</td>
<td>5</td>
<td>1152</td>
<td>99.30</td>
<td>99.77</td>
<td>12614</td>
</tr>
<tr>
<td>80C51N</td>
<td>24204</td>
<td>159</td>
<td>24018</td>
<td>27</td>
<td>9</td>
<td>0.66</td>
<td>99.9</td>
<td>22133</td>
</tr>
</tbody>
</table>

The results also show, that the full-scan versions of the modules are very testable: The numbers of untestable faults in the modules are very low; the fault coverages are almost equal to the (corresponding) fault efficiencies. In addition, the test generation times are very low.

### 8.4 Test generation in non-scan mode

In this section, the experimental results of STPG for the non-scan version of the 80C51 are presented. In addition, also the results of the non-scan versions of the modules (see Table 8.2) of the 80C51 are presented. Table 8.4 shows the results. Circuit WDT(2) is equal to circuit WDT; the STPG time-limit and the maximum test sequence length have been increased for WDT(2) (see later in this section). It is clear that the non-scan version
of the 80C51 (80C51\textsuperscript{N}) is untestable: For only 159 of the 24204 faults, tests have been generated; almost all other faults have been proven untestable. The fault coverage (0.66\%) is a lower bound for the fault coverage of STPG performed on partial-scan versions of the 80C51. The reason for this untestability is that in test-mode, the circuit has many FFs, which are not initializable to 0 and/or 1; as a result, a large fraction of the signal lines (±60\%) is uncontrollable to 0 and/or 1, and almost all signal lines (±96\%) are unobservable. This can be explained as follows:

- **Remodeling:** The original 80C51 circuit has been remodeled to make it suitable for DAT test generation. This remodeling has been performed assuming that combinational circuit TPG has to be performed\textsuperscript{5}. The following aspects have been remodeled and may (partly) have caused the low fault coverage of the circuit:

  - The memory (ROM) in the circuit has been removed, because DAT cannot handle memory. In addition, the special module HOST has been removed. The output of the ROM and HOST are (PI) inputs of the 80C51. These inputs have been set uncontrollable (fixed to value $U$)\textsuperscript{6}; logic elements controlling the ROM and HOST have been removed from the circuit.

  The 128b-RAM memory has been remodeled into buffers; i.e., the signal values on the data-inputs are observable on the outputs (transparent mode of the RAM).

  - Two registers, to select the test-mode of the chip, and to control the BIST protocol of the ROM, have been set to fixed values. These registers contain the code for the SCAN test-mode. To clear these registers for sequential ATPG (which does not use the scan-chains, so selecting SCAN-mode in the registers is not useful/correct) has not been possible; the logic elements to control these registers have been removed.

  - A number of "special" IO-buffers has been remodeled.

- **Clocks:** In test-mode one clock controls the FFs. However, during normal mode the FFs are controlled by four clocks, resulting in the different behavior of the circuit.

- **Asynchronous (re)set:** Asynchronous set and reset capabilities of a number of FFs have been removed. To initialize these (scan)FFs the scan-chain should be used. The DAT system can handle asynchronous (re)set(s). However, we have noticed that for the 80C51 (in test-mode), the asynchronous (re)sets of these FFs have a very low impact on the testability of the circuit. In addition, the logic elements, and wiring to control the (re)sets of the FFs have been removed. Therefore, we have decided not to use the (re)set capabilities of these 7 FFs.

\textsuperscript{5}Actually, the circuit is remodeled for the Philips AMSAL [Hap89] CTPG; Via AMSAL, the 80C51 circuit has been converted into DAT-format (The DAT-format is one of the internal gate-level circuit formats of the DAT ATPG system).

\textsuperscript{6}However, the fault coverage after STPG on the 80C51\textsuperscript{N}, without fixed vector (i.e., allowing STPG to control all PIs) was also very low: 0.77\%.
Hence, in test-mode, the FFs of the circuit can only be initialized using (the) scan FFs; in the 80C51\textsuperscript{N} no SFFs are available, making the circuit uninitializable. Therefore, SFFs have to be selected to make the circuit initializable.

The STPG processes on the modules have been limited to 10 hours. For module WDT(2), the STPG has been limited to 20 hours. The test generation results are very diverse: The modules ALU, INT7L2, UART, WDT(2), and TIMER2 are very testable, and high fault coverages and efficiencies have been reached. The results for CORE, CPU, ASF, I2C and WDT are (sometimes very) poor; hence, the FFs of these modules are candidate for the scan-chain (in the 80C51 circuit). These fault coverages form a lower bound for the fault coverages of partial-scan versions of the modules.

The test set sizes of the non-scan versions (of the modules) are larger than those of the full-scan versions (e.g., compare test set sizes of ALU, INT7L2, and TIMER2). However, assuming that the full-scan versions have a single scan-chain, the test (application) times of the full-scan versions are usually much larger than the test-times of the non-scan versions. E.g., test-time TIMER2 (full-scan): 177 + 1 vectors × 52 SFFs = 9256 clock-ticks; test-time TIMER2 (non-scan): 1152 vectors = 1152 clock-ticks.

A second STPG process has been performed on module WDT (WDT(2)), whereby the maximum test sequence length for a single fault has been increased to 8192 vectors\textsuperscript{7}. Because this module has a timer/counter, long (test) sequences were expected to initialize the counter to the desired states. The results show that our expectation was correct: The test sequence length of module WDT(2) is extremely long (50537 vectors); however, the fault coverage increased from 38.16\% (WDT) to 88.58\% (WDT(2)). The longest test sequence which has been generated for one fault was 6152 vectors. To limit the sequence length, it is clear that one or more FFs of the WDT module have to become scannable.

8.5 Test generation in partial-scan mode

This section presents test generation with partial-scan versions of the 80C51. Techniques to select SFFs, which have been applied to the 80C51, are described, and experimental results of STPG on these partial-scan versions are presented.

In the previous two sections, the lower bound (0.66\%) and upper bound (91.8\%) for the fault coverage have been determined. The 80C51\textsuperscript{N} has been shown to be untestable, therefore, SFFs have to be selected to increase the testability of the circuit; i.e., increasing the fault coverage. The number of selected SFFs has to be as low as possible (for reasons which have been mentioned in Section 8.1 and Section 1.3), while the fault coverage has to reach the upper bound.

Much research has been performed on SFF selection [Agr87] [Gup89] [Lee90] [Mot90]; it has resulted in very sophisticated SFF selection methods. However, SFF selection is not within the scope of this thesis’ research; only very little research has been performed on SFF selection. In this section, a number of SFF selection techniques is proposed.

\textsuperscript{7}Usually the maximum test sequence length is 1024 vectors in the most powerful ATPG stages.
### 8.5. Test generation in partial-scan mode

<table>
<thead>
<tr>
<th>Version</th>
<th>Method</th>
<th>#SFF</th>
<th>#NSFF</th>
<th>Fraction(%)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>V0</td>
<td>–</td>
<td>0</td>
<td>807</td>
<td>0</td>
<td>80C51N</td>
</tr>
<tr>
<td>V1</td>
<td>1</td>
<td>19</td>
<td>788</td>
<td>2.4</td>
<td></td>
</tr>
<tr>
<td>V2</td>
<td>2</td>
<td>761</td>
<td>46</td>
<td>94.3</td>
<td>including V1</td>
</tr>
<tr>
<td>V3</td>
<td>3</td>
<td>173</td>
<td>634</td>
<td>21.4</td>
<td>including V7</td>
</tr>
<tr>
<td>V4</td>
<td>3</td>
<td>196</td>
<td>611</td>
<td>24.3</td>
<td>including V7</td>
</tr>
<tr>
<td>V5</td>
<td>4</td>
<td>344</td>
<td>473</td>
<td>42.6</td>
<td>including V3</td>
</tr>
<tr>
<td>V6</td>
<td>4</td>
<td>447</td>
<td>360</td>
<td>55.4</td>
<td>including V5</td>
</tr>
<tr>
<td>V7</td>
<td>6</td>
<td>68</td>
<td>739</td>
<td>8.4</td>
<td>including V1</td>
</tr>
<tr>
<td>V8</td>
<td>3</td>
<td>253</td>
<td>554</td>
<td>31.4</td>
<td>including V7</td>
</tr>
</tbody>
</table>

#SFF: Total number of SFFs  
#NSFF: Total number of non-scan FFs  
fraction: Fraction SFFs of total number of FFs (= 807)

**Table 8.5: Number of selected SFFs after applying a selection method**

Subsection 8.5.1 presents the SFF selection techniques which have been applied to the 80C51. Subsection 8.5.2 presents techniques to reduce the number of selected SFFs; due to dependencies between FFs, sometimes FFs do not have to become SFFs. Table 8.5 shows the number of selected SFFs (#SFF) and the number of remaining non-scan FFs (#NSFF), after applying one or more SFF selection and/or reduction techniques. In the remainder of this section this table will be explained. Finally, in Subsection 8.5.3 the results are presented for a number of partial-scan versions of the 80C51 (versions V3, V4, V5, V6, and V8, shown in Table 8.5), and for the 80C51 modules which has been listed in Table 8.2.

#### 8.5.1 Scan flipflop selection

In this subsection, four methods are described to select SFFs. The first two methods are required to guarantee that the circuit is initializable, and that all FFs can become observable on a PO (i.e., they aim at improving the fault coverage). The pre-process to determine uninitializable and unobservable signal lines (and thus also uninitializable and unobservable FFs) (see Section 5.1.3, Page 81) is applied to give that guarantee. The last two methods select SFFs to increase the testability of the circuit, hence reduce the (time) complexity of STPG (i.e., they (mainly) aim at improving the fault efficiency).

Only these four methods are described (in the next four paragraphs), because mainly these methods have been applied to the 80C51 and its modules. However, more methods do exist and have been applied. For example, selecting SFFs aiming at reducing the sequential depth of the circuit, or selecting SFFs based on GIS learning (see Section 7.2) with one-value-assigned candidate GISes. Many methods have overlap with each other, as could be expected; i.e., they would (partly) select the same SFFs.

**Method 1: Select all FFs which drive removed elements.** FFs which do not drive elements anymore, because these elements have been removed during the remodeling
<table>
<thead>
<tr>
<th>Testability ($v \in {0, 1}$)</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>(TEST_SUM_SA - v) (c)</td>
<td>(MC^v(c) + MO(c))</td>
</tr>
<tr>
<td>(TEST_MUL_SA - v) (c)</td>
<td>((MC^v(c) + 1) \times (MO(c) + 1))</td>
</tr>
</tbody>
</table>

**Table 8.6: Formulas for testability of signal line c**

Phase in AMSAL (see Section 8.4), have to become scannable, to be useful or otherwise they would not be observable. These SFFs are (only) used to observe signal lines; they cannot control signal lines (they drive no elements).

The 80C51 has 19 FFs of this type; they are selected as SFFs (see also Table 8.5, version V1).

**Method 2: Select FFs which are uninitializable or unobservable.** After the preprocess FUN, to identify uninitializable and/or unobservable lines (see Section 5.1.3) has been performed, a list of FFs has been identified which are uninitializable to 0/1 or are unobservable at any PO. These FFs are selected to become scannable.

The 80C51 has 105 FFs which are both uninitializable to 0/1 (i.e., they are fixed to value \(U\)), 106 FFs are uninitializable to 0 and 531 to 1. 37 FFs have been identified as unobservable. As a result, 761 FFs (including the SFFs of V1) have been selected as SFFs (V2 in Table 8.5). Note that the FFs are not independent of each other. Therefore, a (much) more optimal SFF selection is possible, because making one FF scannable may well lead to other FFs becoming controllable/observable as well.

**Method 3: Select FFs which drive many signal lines with a low testability.** The testability of a signal line is related to the SAFs on this line. Four testability values are associated to a signal line, two for each SAF. These values express the (estimated) cost required to activate a fault and propagate this fault to a PO. Table 8.6 shows the formulas to determine those four testability values; \(TEST\_SUM\_SA - v\) for a fault \(SA - v\) on line c, is the summation of the controllability \(MC^v(c)\) and observability \(MO(c)\), and \(TEST\_MUL\_SA - v\) is the multiplication of the controllability and observability\(^8\). A high testability value implies a difficult to test fault (hard testable). Based on these testability values, the SFFs are selected as follows:

**Step 1** For all faults \(SA - v\), which are not removed by collapsing and which are still undetected (i.e., not tested and not proven untestable), with corresponding signal line c, \(TEST\_SUM\_SA - v\) (c) and \(TEST\_MUL\_SA - v\) (c) are calculated, and stored in two testability value lists. These lists are sorted on the testability values, the highest values on top.

---

\(^8\) Multiplication may give a better impression of the testability of a fault, because it incorporates the (exponentially) increased complexity when both fault activation and propagation have to be satisfied. However, the calculated values often become larger than the maximum (32-bit) integer value, especially in sequential circuits.
### 8.5. Test generation in partial-scan mode

<table>
<thead>
<tr>
<th>TEST_ADD</th>
<th>FF-influence</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td>400</td>
</tr>
<tr>
<td>f2</td>
<td>300</td>
</tr>
<tr>
<td>f3</td>
<td>250</td>
</tr>
<tr>
<td>f4</td>
<td>225</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TEST_MUL</th>
<th>FF-influence</th>
</tr>
</thead>
<tbody>
<tr>
<td>f2</td>
<td>10000</td>
</tr>
<tr>
<td>f1</td>
<td>9000</td>
</tr>
<tr>
<td>f5</td>
<td>8000</td>
</tr>
<tr>
<td>f4</td>
<td>7500</td>
</tr>
</tbody>
</table>

**FF numbers:** [7, 2, 3, 4, 5, 6] **FF distribution array**

**TEST_ADD:** [3, 1, 4, 0, 1, 2] **TEST_MUL:** [3, 1, 2, 0, 1, 2] **FF1 & FF3 selected for SFF**

**Figure 8.3:** Example of method 3 for SFF selection

**Step 2** For each list the following is performed: n faults with the highest testability cost are selected. For each of these faults, the set of FFs (within one timeframe) is determined, which can influence the signal line corresponding to that fault; i.e., have impact on the controllability of this line and (in a minor way) the observability of this line. Hence, for each selected fault, a set of FFs is determined; when the fault is selected from both lists, the fault has two sets of influencing FFs.

**Step 3** Two FF distribution arrays (one for each testability value list) are determined, using each of the sets of FFs of the n selected faults, which express how often an FF influences the n selected faults.

**Step 4** The FFs with the highest values in the FF distribution arrays are selected as SFFs. Depending on the distribution of the values in these arrays, fewer or more FFs are selected. For example, if only a few FFs have high values in the array and the remaining FFs have very low values, then only these few FFs are selected. But, if the distribution of values is similar for a lot of FFs, then we apply trial (select a number of FFs for scan (with highest values in the distribution array)) and error (perform STPG, to determine the fault efficiency; if the efficiency increase is low, select more FFs from the array, etc.).

Figure 8.3 shows an example to illustrate this scan selection method. Assume that the faults f1 . . . f4 shown in the figure are the faults with the highest testability costs (step 1). After each fault-testability combination, the sets of FF(number)s are shown which influence that fault (step 2). Based on these sets of FFs, the FF distribution arrays are determined (step 3). FF 1 and FF 3 have the most influence on the faults, hence they are selected as SFF (step 4).

Based on this selection method, three partial-scan versions of the 80C51 have been determined: V3, V4, and V8 in Table 8.5 (Page 179). The versions are based on version V7, which is an “optimal” selection of SFFs to remove the uninitialized and unobservable FFs (see Section 8.5.2). V3 has been determined by applying this method on the 1000 hardest faults, and V4 and V8 are based on the 5000 hardest faults; for V8 a larger set of SFFs has been selected than for V4.

**Method 4: Select FFs in modules with a low fault efficiency, after STPG.** Based on bad STPG results for a number of 80C51 modules (see Section 8.4), SFFs are selected

---

9 The complete set of FFs which influence the observability of a signal line is much larger
from those modules. In addition, also SFFs of modules are selected when the partial fault efficiency of the module, inside the 80C51, is low; i.e., many faults inside that module have been aborted during test generation for (a version of) the 80C51.

To improve the (partial) fault efficiency of a module (in the 80C51), a number of SFFs has to be selected. We propose to initially select all FFs of a module in the 80C51. After that (if STPG results have been successful), techniques to reduce the number of selected SFFs can be applied (see next subsection). For example, the fault efficiency of module I2C is very low (see Table 8.4), therefore, the FFs of module I2C have been selected as SFF, in the 80C51.

Based on this selection method, two partial-scan versions of the 80C51 have been derived (see Table 8.5):

V5: which includes the SFFs of V3, supplemented with the FFs of modules I2C, UART, WDT, and XRAMIF. These modules have been added because: The non-scan version of the I2C module has a low fault efficiency; in the UART and the XRAMIF module, many faults have been aborted after STPG of the 80C51.V3 (80C51 with V3 SFF selection); the non-scan version of the WDT module has very long test sequences.

V6: which includes the SFFs of V5, increased with FFs of modules TIMER2 and TIMER1. STPG on the non-scan versions of these modules performs very good. However, after STPG of the 80C51.V3 many faults have been aborted inside these modules.

### 8.5.2 Scan flipflop reduction

Usually, in a sequential circuit, many FFs depend on other FFs; i.e., FFs are driven by other FFs. So, often when a number of SFFs has been selected, not all of these FFs have to become scannable. Figure 8.4a shows an example circuit, with two uninitializable FFs. Scan selection method 2 would select both FFs. However, it is easy to see that $FF_1$ and $FF_2$ depend on each other; when $FF_2$ becomes an SFF, also $FF_1$ becomes initializable.

In the next two paragraphs techniques are described to reduce the number of selected SFFs. The first paragraph describes a technique to reduce the number of (selected) SFFs based on dependency graphs/matrices, which express dependencies between (S)FFs and
8.5. Test generation in partial-scan mode

![Dependency graph of 5 FFs, within one timeframe; b) corresponding matrix; c) dependency matrix over timeframes](image)

Figure 8.5: a) Dependency graph of 5 FFs, within one timeframe; b) corresponding matrix; c) dependency matrix over timeframes

can identify closely related FF groups. The last paragraph presents SFF reduction techniques based on visual inspection of the circuit, and SFF reduction based on “trial and error” in (S)FF groups.

Reducing the number of SFFs based on dependency graphs/matrices. Directed dependency graphs and/or corresponding matrices can be generated for circuits to visualize the dependencies between inputs, outputs, and FFs. Figure 8.4b and Figure 8.5a show two dependency graphs. The nodes represent the FFs and the directed arcs express the dependencies. E.g., the directed arc in Figure 8.5a from node d to node a expresses that FF_a is dependent of FF_d. The same information is also stored in the matrix of Figure 8.5b. The dependency between FF_a and FF_d is stored in entry (a, d). A loop in the graph denotes that the FFs in the loop are dependent of each other; e.g., FF_a, FF_c, and FF_d (see Figure 8.5a) form a loop and thus are dependent of each other.

Assume that the FFs in the graph are uninitializable. To make the FFs initializable, the loops have (at least) to be broken (because the FFs are uninitializable due to these loops) by selecting one or more FFs in the loops as SFF. E.g., selecting FF_2 (in Figure 8.4b) to be a SFF will break the loop, and probably makes all the FFs initializable. We propose the following (heuristic) rules, to be applied for SFF selection based on the dependency graph and/or matrix, and which will break one or more loops:

1. Select all FFs which have a self-loop. E.g., FF_b in Figure 8.5a. The self-loop can be deterministically identified in the dependency matrix: The 1’s on the diagonal of the matrix corresponds to the self-loops in the graph (see Figure 8.5b).

2. Select the FFs which have the highest connectivity. E.g., FF_c in Figure 8.5a has a connectivity of 5, which is the highest one for this graph. Therefore, FF_c is selected as SFF. This connectivity can be determined by summation of the rows and columns of the matrix of Figure 8.5b. E.g., the connectivity of FF_c is the sum of row c (= 2) and the sum of column c (= 3), which is a total connectivity of 5.

The SFF selections may be not optimal; however, the number of selected FFs is reduced compared to the selection methods of the previous subsection. Also, it is not guaranteed that the selected SFF makes the other FFs in the loop(s) (consisting this selected SFF) ini-
Figure 8.6: b) \( FF_1 \), which depends on itself, can be initialized

tializable. This has to be determined with the pre-process FUN to identify uninitializable FFs (see Section 5.1.3).

Based on the dependency matrix, also FF groups can be identified. These FFs are all dependent of each other. Figure 8.5c shows that all (S)FFs of the dependency graph in Figure 8.5a are dependent of each other, hence, they form an FF group. This matrix is determined by repeatedly “adding” a row \( R \) of the matrix in Figure 8.5b, to another row, which has an assigned entry on column \( R \). E.g., row \( a \) has an assigned entry on column \( d \); hence, row \( d \) is added to row \( a \). Continuing this process, the matrix of Figure 8.5c results. This process continues until no changes are detected anymore in the matrix of Figure 8.5c. In larger circuits, multiple FF groups may be identified. Within an FF group, a minimal number of SFFs can be selected, so that all FFs in the group become initializable (see next paragraph).

Selecting SFFs by visual inspection of the dependency graph can only be performed for small circuits; e.g., modules XRAMP, PCON and XRAMIF. Selecting SFFs using the dependency matrix can be applied for much larger circuits; however, the above described rules are not optimal, so sometimes too few or too many SFFs are selected. Therefore, the proposed solution based on this reduction method is often optimized using visual inspection of the circuit, and applying “trial and error” on SFF selection.

Reducing the number of SFFs based on visual inspection in the circuit, and “trial and error”. Determining dependency graphs/matrices results in a number of SFFs being selected. However, how many SFFs have to be selected? Which SFFs are necessary/required to be selected? This information is not stored in the graphs/matrices. Therefore, in addition, we have applied visual inspection in the circuit. Visual inspection in the circuit, to determine “real” dependencies between FFs has been successful in reducing the number of SFFs. For example, Figure 8.4a shows a circuit with two uninitializable FFs. Based on the dependency graph of this circuit, \( FF_1 \) or \( FF_2 \) can be selected as SFF. However, it is easy to find out that only selecting \( FF_2 \) will make both FFs initializable. A dependency graph/matrix can not identify this, because it does not store the logic value propagation information between two FFs. E.g., consider Figure 8.4a again, value 0 and 1 in \( FF_2 \) influence the state of \( FF_1 \); value 0 and 1 in \( FF_1 \) can not influence the state of \( FF_2 \); hence, using the value propagation information, \( FF_2 \) is not dependent of \( FF_1 \), and \( FF_2 \) have to be selected as SFF. Thus FFs can be uninitializable due to loops and/or the impossibility to propagate (Boolean) logic values to these FFs.

Based on the dependency graphs, self-looping FFs are selected to become SFF. However, Figure 8.6 shows an example of a self-looping FF, which can be initialized. This
situation often occurs in the 80C51 circuit. Using visual inspection many of these initial-
izable self-loops have been identified, saving on selected SFFs based selection method
which use dependency graphs (see previous paragraph).

It is clear, that the two examples explained above, can also be performed automatic-
ically. However, in larger circuits (such as the 80C51) logic value propagation, to find
out "real" dependencies, is much more (time) complex, because the propagation of values
depends on assignments of other FFs and PIs.

Another method we have applied to reduce the number of SFFs, is *trial and error* on
the selection of SFFs in FF groups (which has been identified by dependency graphs/matrices).
Only applying the two rules, defined in the previous paragraph, may result in that
FFs are still not initializable and/or unobservable. Selecting additional SFFs or removing
SFFs (i.e., by making FFs unscannable again), using the circuit topological information
and dependency information of FFs, may result in more optimal sets of SFFs (*trial*).
The successfullness of a new set of SFFs is checked by performing the pre-process FUN
to identify uninitializable/unobservable FFs, and by performing STPG to determine the
reached fault coverage/efficiency. If the results are not satisfactory (*error*), a new selec-
tion is performed.

Experiments on the 80C51 circuit indicate that none of the proposed techniques to reduce
the SFFs has been successful on its own. To identify an optimal set of SFFs all techniques
have to be applied. The SFF-set of the 80C51_V2 (see Table 8.5, Page 179), which has
761 SFFs, is reduced to a SFF-set of only 68 SFFs when doing this (*V7*). Based on this
small set of SFFs, all uninitializable and unobservable FFs have been removed. However,
the fault coverage and efficiency for circuit *V7* is still very low, and therefore additional
SFFs have been selected based on methods 3 and 4 (see Subsection 8.5.1), resulting in
versions *V3, V4, V5, V6 and V8*.

### 8.5.3 Experimental results

Experiments have been performed on partial-scan versions of the modules (see Table 8.2)
of the 80C51, and the following five partial-scan versions of the 80C51: *V3, V4, V5, V6,
and V8* (see Table 8.5, Page 179). These partial-scan versions of the 80C51 are denoted as
80C51_V3, 80C51_V4, 80C51_V5, 80C51_V6, and 80C51_V8, respectively. The partial-
scan versions of the modules have been determined by applying method 3; a number of
SFFs has been selected based on the 100 hardest faults. Table 8.7 shows the results of
ATPG for the partial-scan versions. For a number of modules two sets of SFFs has been
determined (using method 3). Column 1 shows the name of these modules together with a
version number; e.g., CORE(1) is module CORE with the first set of SFFs, and CORE(2)
is also module CORE, however with the second set of SFFs. Column 2 shows the number
of selected SFFs and the total number of FFs. Table 8.8 shows the fault coverages and
the test set size for the non-scan, partial-scan and full-scan versions of the modules of the
80C51. The ATPG time limit for the modules has been set to 10 hours and for the 80C51
to 100 hours.
<table>
<thead>
<tr>
<th>Name</th>
<th>#SFF/#FF</th>
<th>#tst</th>
<th>#unt</th>
<th>#vec</th>
<th>fcover (%)</th>
<th>eff (%)</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORE (1)</td>
<td>52/475</td>
<td>5470</td>
<td>328</td>
<td>183</td>
<td>33.91</td>
<td>36.88</td>
<td>10h</td>
</tr>
<tr>
<td>CORE (2)</td>
<td>107/475</td>
<td>13555</td>
<td>326</td>
<td>3072</td>
<td>84.04</td>
<td>87.17</td>
<td>10h</td>
</tr>
<tr>
<td>CPU</td>
<td>45/184</td>
<td>5833</td>
<td>58</td>
<td>1390</td>
<td>75.64</td>
<td>76.45</td>
<td>10h</td>
</tr>
<tr>
<td>ALU (1)</td>
<td>6/39</td>
<td>2225</td>
<td>23</td>
<td>369</td>
<td>98.84</td>
<td>100.00</td>
<td>736.02s</td>
</tr>
<tr>
<td>ALU (2)</td>
<td>15/39</td>
<td>2225</td>
<td>23</td>
<td>267</td>
<td>98.84</td>
<td>100.00</td>
<td>50.04s</td>
</tr>
<tr>
<td>INTL7L2 (1)</td>
<td>18/91</td>
<td>3328</td>
<td>9</td>
<td>1320</td>
<td>98.03</td>
<td>99.06</td>
<td>10h</td>
</tr>
<tr>
<td>INTL7L2 (2)</td>
<td>40/91</td>
<td>3338</td>
<td>11</td>
<td>1264</td>
<td>98.32</td>
<td>99.26</td>
<td>10h</td>
</tr>
<tr>
<td>ASF (1)</td>
<td>25/314</td>
<td>6225</td>
<td>18</td>
<td>1667</td>
<td>74.93</td>
<td>75.28</td>
<td>10h</td>
</tr>
<tr>
<td>ASF (2)</td>
<td>64/314</td>
<td>8199</td>
<td>18</td>
<td>6477</td>
<td>98.69</td>
<td>99.49</td>
<td>10h</td>
</tr>
<tr>
<td>UART</td>
<td>13/65</td>
<td>1302</td>
<td>0</td>
<td>1247</td>
<td>96.52</td>
<td>98.30</td>
<td>4.11h</td>
</tr>
<tr>
<td>I2C (1)</td>
<td>9/65</td>
<td>1794</td>
<td>0</td>
<td>1368</td>
<td>88.72</td>
<td>89.56</td>
<td>10h</td>
</tr>
<tr>
<td>I2C (2)</td>
<td>24/65</td>
<td>1955</td>
<td>0</td>
<td>1178</td>
<td>96.69</td>
<td>97.48</td>
<td>5.82h</td>
</tr>
<tr>
<td>WDT</td>
<td>7/29</td>
<td>554</td>
<td>0</td>
<td>2112</td>
<td>94.38</td>
<td>95.91</td>
<td>3.51h</td>
</tr>
<tr>
<td>TIMER2</td>
<td>8/52</td>
<td>2122</td>
<td>1</td>
<td>1014</td>
<td>99.62</td>
<td>99.86</td>
<td>123.8m</td>
</tr>
</tbody>
</table>

| 80C51.V3  | 173/807 | 17501 | 2132 | 4571 | 72.3      | 81.1    | 98h   |
| 80C51.V4  | 196/807 | 18557 | 2133 | 5336 | 76.7      | 85.5    | 82h   |
| 80C51.V8  | 253/807 | 19441 | 2074 | 6878 | 81.0      | 89.5    | 100h  |
| 80C51.V5  | 344/807 | 20734 | 2114 | 8003 | 85.7      | 94.4    | 100h  |
| 80C51.V6  | 447/807 | 21778 | 2102 | 7589 | 90.0      | 98.7    | 37h   |

Table 8.7: Experimental results 80C51 circuit and 80C51 modules (partial-scan)

The fault efficiencies and coverages reached for the partial-scan versions of the modules are very high, compared to the non-scan versions. Furthermore, the computing times have been reduced. For almost all modules, the fault coverages of the partial-scan versions reach nearly the fault coverage of the full-scan versions. For modules, for which the fault coverages for non-scan, partial-scan, and full-scan is similar, the number of generated vectors is highest for the non-scan versions and lowest for the full-scan version. However, assuming that the modules have a single scan-chain, the expected test-time for the non-scan and/or partial-scan versions often are lower than for the full-scan versions. E.g., the test-time for the UART: Non-scan: 1614 clock-ticks, partial-scan: \((1247 + 1) * 13 = 16211\) clock-ticks, and full-scan: \((69 + 1) * 65 = 4550\) clock-ticks. The test-time for the ALU: Non-scan: 509 clock-ticks, partial-scan: \((6 + 1) * 369 = 2583\) clock-ticks, and full-scan: \((39 + 1) * 135 = 5400\) clock-ticks. The test-time for the INTL7L2: Non-scan: 1983 clock-ticks, partial-scan: \((18 + 1) * 1320 = 25080\) clock-ticks, and full-scan: \((91 + 1) * 186 = 17112\) clock-ticks. Note that for all versions very little effort has been spent in reducing the test set size; i.e., (static and/or dynamic) test set compaction techniques have not been applied, except for the full-scan versions where reverse fault-simulation has been performed after the test set was generated. This may explain the sometimes lower test-time for full-scan versions of the modules, compared to partial-scan versions.

The results for the partial-scan versions of the 80C51 show that a reasonable number of SFFs has to be selected to achieve a high fault coverage/efficiency (see results of 80C51.V5 and 80C51.V6 in Table 8.7). The test-time of the circuit is probably reduced, compared to the test-time of the full-scan version. However, the number of generated test
8.5. Test generation in partial-scan mode

<table>
<thead>
<tr>
<th>Name</th>
<th>Non-scan</th>
<th>Partial-scan</th>
<th>Full-scan</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fcov(%)</td>
<td>#vec</td>
<td>#SFF/FF</td>
</tr>
<tr>
<td>CORE(1)</td>
<td>13.90</td>
<td>21</td>
<td>52/475</td>
</tr>
<tr>
<td>CORE(2)</td>
<td>13.90</td>
<td>21</td>
<td>107/475</td>
</tr>
<tr>
<td>CPU</td>
<td>21.97</td>
<td>42</td>
<td>45/184</td>
</tr>
<tr>
<td>ALU(1)</td>
<td>98.76</td>
<td>509</td>
<td>6/39</td>
</tr>
<tr>
<td>ALU(2)</td>
<td>98.76</td>
<td>509</td>
<td>15/39</td>
</tr>
<tr>
<td>INT7L2(1)</td>
<td>98.00</td>
<td>1983</td>
<td>18/91</td>
</tr>
<tr>
<td>INT7L2(2)</td>
<td>98.00</td>
<td>1983</td>
<td>40/91</td>
</tr>
<tr>
<td>ASF(1)</td>
<td>62.83</td>
<td>1063</td>
<td>25/314</td>
</tr>
<tr>
<td>ASF(2)</td>
<td>62.83</td>
<td>1063</td>
<td>64/314</td>
</tr>
<tr>
<td>UART</td>
<td>82.21</td>
<td>1614</td>
<td>13/65</td>
</tr>
<tr>
<td>I2C(1)</td>
<td>50.79</td>
<td>194</td>
<td>9/65</td>
</tr>
<tr>
<td>I2C(2)</td>
<td>50.79</td>
<td>194</td>
<td>24/65</td>
</tr>
<tr>
<td>WDT</td>
<td>38.16</td>
<td>58</td>
<td>7/29</td>
</tr>
<tr>
<td>TIMER2</td>
<td>99.30</td>
<td>1152</td>
<td>8/52</td>
</tr>
</tbody>
</table>

Table 8.8: Comparison of ATPG results with non-scan, partial-scan and full scan versions of the modules

Figure 8.7: Relation between fraction of SFFs and fault coverage/efficiency (after 100 hours TPG)

Vectors increases when the number of SFFs increases; an explanation for this (perhaps unexpected) behavior may be the increased fault coverage. In addition, no effort has been spent in reducing the test set size. Figure 8.7 shows the relation between the fault coverage/efficiency and the fraction of selected SFFs. Based on this figure, only 50% percent of FFs have to be scannable in order to reach a similar fault coverage as for the fault coverage of the 80C51F.

For all five partial-scan versions (V3, V4, V5, V6, and V8), the difference between the fault coverage and fault efficiency is somewhere between 8%—10% (see Figure 8.7). This value is comparable to the difference in coverage and efficiency of the full-scan version. This implies that, with the selected SFFs, the fault coverage upper-bound (91.8%) can be very closely reached, when computing time is unlimited; i.e., the number of untestable faults is not increased due to non-scan FFs.

The number of selected SFFs of version V5 and V6 can be reduced when not all FFs (method 4) of a hard-to-test module are selected, but a selected number. E.g., the non-
Chapter 8. Case study: ATPG for the Philips 80C51

The scan version of module I2C has a poor fault coverage (45%); selecting 24 SFFs (36.9%) improves the fault coverage to almost 98% (see Table 8.7, circuit I2C(2)); hence, probably only these FFs have to be selected in version V5 and V6. In addition, the reduction techniques of Subsection 8.5.2 can be applied to the FFs of the selected modules and to the SFFs of the 80C51 versions. But a reduced number of SFFs may increase the complexity of STPG, and thus reduce the reached fault coverage after 100 hours of computing time (Figure 8.7 seems to support this impression). However, note the following: Assume a SFF set, called version V6a, which is an optimal set of SFFs (using the reduction techniques) of the SFF set of V6, and that the sizes of the SFF sets of V6a and V4 are similar. The expected fault coverage for V6a is higher than for V4 because dependencies between FFs have been removed in V6a (i.e., V6 has no, or less redundant SFFs than V4 has; only essential SFFs are selected in V6a, because the reduction techniques have been applied for set V6a and not for V4). However, manually performing the reduction techniques is a time consuming task, and therefore, these techniques have to be automated.

The computing times required to reach the fault efficiencies are extremely high. However, usually with relative low CPU costs, almost the final fault efficiency has been reached (because the first ATPG stages have low backtrack limits); the remaining time has been spend for the last few percents of fault efficiency (in the ATPG stages with higher backtrack limits). Hence, a few percent lower fault efficiency results in a very large reduction of CPU costs. In addition, all runs have been performed on development versions of the DAT system, which contain a lot debugging and checking code. Lower computing times (but probably still very large) are to be expected when an optimized version is applied.

The memory consumption of STPG has been sometimes very large; some versions of the 80C51 have required more than 1GB of memory, in spite of the memory reduction techniques which have been applied in DAT (see Section 7.4). However, this occurs only for the hard faults, for which the backtrack-limits and timeframe-limits have been set very high.

8.6 Summary and conclusions

In this chapter, a case study has been presented to demonstrate the effectiveness and efficiency of the sequential circuit ATPG system DAT. The Philips micro-controller 80C51 has been the subject of our study. ATPG has been performed on a full-scan, a non-scan, and several partial-scan versions of the 80C51. Because design departments within Philips are interested in synthesizing the 80C51 core without scan or with partial-scan, an additional goal of this study is to select an as small as possible set of SFFs, while maintaining the full-scan fault coverage.

For the full-scan version of the 80C51 a fault coverage of 91.8% has been reached, and for the non-scan version a fault coverage of only 0.66% has been reached. These coverages form the upper and lower bound for the fault coverages of the partial-scan versions.

To perform SFF selection, a number of methods has been proposed and described. The methods select SFFs based on uninitializeable, unobservable, hard-to-control, and/or
hard-to-observe signal lines. In addition, SFF are selected based on the impact the FFs have on aborted faults, and based on STPG results of modules of the 80C51. In order to reduce the number of selected SFFs, techniques have been proposed based on dependency graphs/matrices, visual inspection of the circuit topology, and on “trial and error” on groups of SFFs.

ATPG has been performed on five partial-scan versions of the 80C51. The SFFs have been selected using the methods and techniques proposed in this chapter. The number of SFFs ranges from 21.4% up to 55.4% of the total number of FFs. The obtained fault coverages range from 72% up to 90%; which is almost equal to the fault coverage reached for the full-scan version. The CPU times required to reach those coverages are very high (range between 37 and 100 hours). The results demonstrate that a reasonable number of FFs (±50%) has to be scannable for the 80C51, to reach a fault coverage in test generation similar to the full-scan version. The fault coverage is reduced by ±10% percent, when ±30% of the FFs have been selected for scan.

Comparing the results of the 80C51 to the results of the ISCAS'89 circuits (see Chapter 6 and 7) shows that the 80C51 is on the border of what state-of-the-art STPGs can handle; larger circuits than the 80C51 (e.g., the non-scan ISCAS’89 circuits s38417, s38584) could not (yet) be successfully handled by STPGs.

Concluding this chapter, based on the results for partial-scan versions of the 80C51, the ATPG system DAT performs satisfactory. In spite of the huge complexity of STPG, DAT reaches high fault coverages, especially when the number of SFFs of the 80C51 is more than 40%. However, the results also show the limitations of the system; there is still work to do in improving the STPG system.

Considering SFF selection for partial-scan, many of the proposed techniques have to be performed automatically; i.e., have to be implemented, to achieve better SFF selections.
Chapter 9

Conclusions

In this final chapter, the main subjects of this thesis are summarized, the main contributions are enumerated (Section 9.1), and future research (Section 9.2) is proposed.

As has been described and motivated in Chapter 1, structural testing is a minimal requirement for guaranteeing that an IC is free of defects, hence increasing the likelihood of a working IC. Therefore, ATPG is a necessary part in the design flow of an IC. In this thesis, ATPG for one type of IC has been investigated: synchronous sequential circuits. This has been motivated by the fact that ATPG for sequential circuits has not been solved: The fault efficiencies for the circuits beyond the very small ones are often very poor, and/or the test generation times are very long. The use of the scan design methodology, which can reduce the test generation effort for sequential circuits to the effort required for a combinational circuit, results in area, performance and test-time penalties, and is therefore often not (fully) applicable. This has further motivated this research.

The overall target has been defined as: The development of an ATPG system for sequential circuits which reaches higher fault efficiencies and/or reduces test generation times in comparison to results published by other state-of-the-art ATPG systems [Nie91] [Gou93] [Dar97]. To reach this target, the DAT ATPG system for combinational circuits [vdL96] has been modified and extended to be able to handle 3-state synchronous sequential circuits. Moreover, new techniques and concepts have been added to the system to reduce the complexity of TPG for sequential circuits (STPG). However, before these modifications, extensions, and additions to DAT have been proposed (in Chapter 4) and have been described (in Chapters 5, 6, and 7), in Chapters 2 and 3 background and required terminology in the field of sequential circuits and test generation have been given.

In Chapter 2 (Circuit, signal, and fault models), the synchronous sequential circuit model, the 5V signal model and the SAF fault model, used throughout this thesis, have been described. Also, the iterative array model has been introduced. This model makes the usage of combinational test generation algorithms for sequential circuits possible.

In Chapter 3 (ATPG system structure), the ATPG system structure has been presented. The main components of an ATPG system have been identified and presented. These components are the pre-processes, the simulation-based TPGs, the deterministic TPGs, the fault simulators, the learning processes, and the post-processes. A detailed descrip-
tion of the FAN-algorithm for combinational circuits has been given. The relevant methods and techniques applied in the FAN-based TPG DAT [vdL96] have been introduced: The 2SV signal model, three-valued decisions and objectives, bus-conflict detection and bus-padding, and dynamic dominators. In the last part of this chapter, existing STPG methods/algorithms have been categorized into several STPG categorizations, and shortcomings, advantages or disadvantages of these methods have been studied. It has been made clear that current state-of-the-art (deterministic) STPG methods are based on the FAN algorithm and the CTP approach.

In Chapter 4 (Problems in STPG and proposed solutions), three problems have been identified which (can) occur in TPG for sequential (3-state) circuits, and can invalidate the correctness and the successfulness of the STPG: First, due to shortcomings of the applied signal model, the STPG can be pessimistic in bus-conflict detection. Second, due to shortcomings of the applied signal model and the decision-making process, the search of the STPG can be incomplete, or can result in over-specified states, or can be inefficient in terms of number of backtracks. This may result in too high CPU costs, needlessly longer test-sequences, or even in faults being incorrectly marked untestable. Third, detection and storage of the illegal state space is essential for a successful STPG. However, often a small part of the illegal state space has been identified during STPG, and the identified ISes are usually over-specified. This over-specification may result in (too) large IS lists, and (needlessly) reduced knowledge of the IS space.

Prior art related to these problems has been studied. We have found no signal model that simultaneously assures correct search in STPG, prevents over-specified states during test generation, and performs correct bus-conflict detection. Also, very little work has been published on IS detection, storage and optimization, before or during STPG. No practical techniques have been reported which explicitly identify the Global Illegal State (GIS) space.

Four state-of-the-art STPG methods have been investigated: HITEC, DUST, FOBUST and MOSAIC. Shortcomings, disadvantages, but also useful techniques from these STPG methods have been described.

Based on the identified unsolved problems and previous work in STPG, new TPG techniques and processes in an ATPG system for sequential circuits have been proposed. The STPG part of this system applies the CTP approach and is based on the combinational TPGs of DAT. A new power-set based signal model has been proposed to overcome two of the three identified problems: Incorrect bus-conflict detection, and incomplete and inefficient search in TPG. Finally, techniques and processes have been proposed to solve the third identified problem: Illegal state space detection and over-specified identified ISes.

In Chapter 5 (ATPG for sequential circuits), a state-of-the-art ATPG system for sequential 3-state circuits has been presented. A three-phase STPG has been described. Phase 1 considers the fault activation, and propagation in timeframe 1. Phase 2 considers the fault propagation in timeframe 2 and further until a PO has been reached. Phase 3 considers the state justification backward in time, until the current state covers the last state of the previously generated test sequence, or the 'all unknown' initial state. The
STPG algorithm is based on the CTPG algorithm in DAT [vdL96], which in turn is based on the FAN algorithm [Fuj83], and incorporates techniques from HITEC [Nie91]. Many aspects of this CTPG algorithm have been modified to handle sequential (3-state) circuits. In addition, new techniques have been added to improve the performance of the STPG: Fault dependent cost estimates, complete and local fault-cones, the state restrictor to constrain the state space, and two techniques to perform some form of back-jumping. Eight STPG (search) strategies have been proposed to improve the performance for classes of sequential circuits.

To support the STPG, two static learning techniques have been proposed: To find global implications, a learning process has been proposed using recursive learning and the logic rules of constructive dilemma and contraposition. Experimental results show an increased number of learned global implications, and/or reduced CPU costs compared to previous published methods. To find signal lines that are uninitializable to 0 and/or 1, a learning process has been proposed that tries to justify signal lines to value 0 and/or 1. If that fails, a new signal line which is uninitializable to 0 and/or 1 is identified.

Two additional target STPG methods have been proposed: The first one extends the current test sequence, by selecting additional targets making use of the fault-effects on the PPO state of the last generated timeframe (method A1). The STPG generates additional timeframes to propagate these fault-effects to a PO. The second one extends the current test sequence, after STPG for a primary target, using the unassigned PIs which are often available in the generated sequences (method A3). The STPG tries to assign these PIs to detect additional faults. Experimental results show that method A1 increases the fault efficiency for almost all ISCAS’89 circuits (the increase ranges from 0.25% up to 5%), while the CPU costs did not change. Methods A3 is very successful for deep sequential circuits (i.e., having more than 50 circuit levels): fault efficiency improvements up to 20% has been achieved. In addition, the test set sizes have been reduced significantly for these circuits.

Finally, all strategies, methods, and techniques, which have been proposed in this chapter, have been combined into an ATPG process. This ATPG process is the basis for the proposals and experiments described in chapters 6, 7 and 8.

In Chapter 6 (ATPG based on Power-Set Logic), a novel signal model, called PSL, based on the power-set of the set of basic values \( \{0, 1, Z\} \) has been presented. PSL based TPG employs three valued decision options to guarantee completeness of search during STPG. In addition, PSL guarantees efficient decision-making during STPG, and avoids over-specified circuit states, because of its ability to express the exclusion of a certain value; as a result, all specified values (value 0 and 1) on PPIs are actually assigned by the backtrace procedure (i.e., these values contribute to the test). It assures that only essential values on the PPIs are justified, reducing STPG effort and leading to more compact test sets. The pessimism in bus conflict detection of most other methods is absent from PSL, so it can be called exact within the realm of non-symbolic reasoning techniques. The value resolution is very high; as a result, much more implications can be performed in the circuit, compared to other signal models. Eight advantages of PSL based TPG over TPG based on enumerative and/or conventional set-based signal models (see Section 6.3 and
Section 6.4) prove the superiority of PSL.

Experimental results confirm this by reaching higher fault efficiencies for industrial circuits as well as for the sequential ISCAS'89 circuits, often combined with a large reduction of computing time. The average reached fault efficiency for the combinational industrial circuits has been increased from 98.8% (25V based CTPG) to 99% (PSL based CTPG), while the CPU costs have been reduced with 15%. The averagely reached fault efficiency for the sequential circuits has been increased from 80.5% (25V based STPG) to 88.3% (PSL based STPG), while the CPU costs have been reduced with 30%.

In Chapter 7 (Illegal state space identification), an ATPG system for sequential circuits has been presented; it focuses on discovering the Global Illegal State (GIS) space, which constrains the search space during STPG to justifiable states. The GIS space is gradually discovered implicitly during STPG for faults. However, these "accidentally" found GISes are often over-specified; i.e., are not prime. Therefore, six techniques have been proposed to expand the known GIS space by trying to remove the over-specification. By using the current GIS list, they generate Potential GISes (PGISes) based on the concepts of GIS identification, merging and expansion. These PGISes are tried to be justified using the Justifier (the Justifier is phase 3 of the STPG (the backward justification phase)). A justification failure for a PGIS results in a new GIS, which usually covers one or more GISes from the original GIS list. This reduces the size of the GIS list (thus saves in memory usage) and expands the known GIS space (thus increases the performance of the STPG).

To improve the comparison speed of two states, FF-groups and cvectors have been proposed. Cvectors also reduce the storage requirements for the ISes up to a factor 10.

The process GIS-learning has been proposed to find sparsely assigned GISes (which are often prime). Experimental results show that GIS-learning often can identify a very large part (or all) of the GIS space, within low computing times. The low computing times are mainly due to the legal state cache, which has been proposed to speed-up the learning process and the optimization techniques.

The experimental results of test generation demonstrate that the proposed techniques are very effective: 100% fault efficiency for a large number of the ISCAS'89 and 3-state industrial circuits has been reached within reasonable CPU time; for 20 out of the 25 first ISCAS'89 circuits 100% fault efficiency have been reached. Compared to other ATPG systems (HITEC, MOSAIC and DUST) usually (much) better results in fault efficiency and/or CPU usage have been obtained.

However, the usefulness of a particular technique is highly circuit dependent. Therefore, it would be desirable to make the selection of the techniques circuit dependent.

In Chapter 8 (Case study: ATPG for the Philips 80C51), a case study has been presented to demonstrate the effectiveness and efficiency of the sequential circuit ATPG system DAT. The Philips micro-controller 80C51 has been the subject of study. ATPG has been performed on a full-scan, a non-scan, and several partial-scan versions of the 80C51. Because design departments within Philips are interested in synthesizing the 80C51 core without scan or with partial-scan, an additional goal of this study was to select an as small
as possible set of SFFs, while maintaining the full-scan fault coverage.

Experimental results show a very high fault coverage for the full-scan version (91.5%) of the 80C51, and almost no fault coverage (0.66%) for the non-scan version. These coverages form the upper and lower bound for the fault coverages of the partial-scan versions.

ATPG has been performed on five partial-scan versions. The SFFs have been selected using the methods and techniques proposed in this chapter. The number of SFFs ranges from 21.4% up to 55.4% of the total number of FFs. The obtained fault coverages range from 72% up to 90%, which is nearly the fault coverage reached for the full-scan version. The CPU times required to reach those coverages are high. The results demonstrate that ±50% of the FFs have to be scannable for a fault coverage comparable to the full-scan version. The fault coverage is reduced by ±10% percent when only ±30% of the FFs have been selected for scan.

Comparing the results of the 80C51 to the results of the ISCAS’89 circuits (see Chapter 6 and 7) shows that the 80C51 is on the border of what state-of-the-art STPGs can handle; larger circuits than the 80C51 (e.g., the non-scan ISCAS’89 circuits s38417, s38584) could not (yet) be successfully handled by our STPG method, but also not by other STPG methods such as HITEC, MOSAIC and DUST.

9.1 Major contributions

The major contributions of this research are the following:

- An ATPG system has been developed, based on the combinational circuit ATPG system DAT, which has been modified and extended to handle (3-state) synchronous sequential circuits. The following modifications and extensions in the ATPG system have been proposed and described:

  - Combinational and sequential controllability cost estimates, and observability cost estimates of signal lines, have been merged.

  - Pre-identification of untestable faults, based on uninitializable and/or unobservable faults.

  - A three-phase STPG, based on the FAN algorithm and applying the CTP approach. The following new techniques have been added to this STPG to improve its performance: Fault dependent cost estimates, backtrace with six counters, three-valued decisions for the fault-free and faulty circuit, the State Restrictor, and two forms of back-jumping.

  - Various (search) strategies during STPG.

- An new signal model has been introduced: The Power-Set Logic signal model (PSL), to improve performance of (S)TPG, to guarantee complete search during (S)TPG, and to avoid pessimistic bus-conflict detection.
• New techniques have been proposed to expand the known (Global) Illegal State (G)IS space, which has been identified during STPG and has been stored in the State Restrictor. These techniques are based on (G)IS identification, merging, and expansion, and aim at identifying a larger (G)IS space.

• Two forms of additional target TPG for sequential circuits have been described.

• Static learning, based on logic implication, has been extended: During the learning process, also recursive learning is performed.

• Static learning, based on signal line justification has been described.

• Static learning, based on IS justification has been proposed (GIS learning).

• The legal state cache has been introduced, to accelerate the justification process of candidate ISes; i.e., to accelerate the process of (G)IS space identification.

• Four methods have been proposed to perform scan-FF selection, and various techniques have been described to reduce the number of selected SFFs.

Experimental results have demonstrated that our ATPG system can handle 3-state synchronous sequential circuits, with or without applying the scan methodology, while respecting all internal and external restrictions. The performance of this ATPG system has been considerably improved (i.e., (much) higher fault-efficiencies and/or lower computing times), compared to other state-of-the-art ATPG systems (see Table 7.4 on Page 168). 100% fault efficiency for a large number of the ISCAS’89 (20 out of the first 25 circuits) and 3-state industrial circuits within reasonable CPU times has been reached. For only 6 out of the 38 ISCAS’89 circuits, the reached fault efficiency are below 90%. Also, the reported fault coverages/efficiencies are correct; i.e., no pessimistic nor optimistic fault coverages/efficiencies have been reported.

9.2 Suggestions for future research

A powerful ATPG system for sequential circuits has been described in this thesis. However, as the experimental results have demonstrated, even with this ATPG system, the problem of sequential circuit ATPG has not been solved completely. However, major steps have been made in the direction of the solution. We suggest the following topics to be investigated, to further improve sequential circuit ATPG:

• Perform simulation-based STPG before deterministic STPG. As we already have indicated in the thesis, a simulation-based STPG can identify many easily detectable faults, but also faults which have been identified to be very hard-testable for deterministic STPG.

• Two forms of (conflict-directed) back-jump techniques during backtracking have been proposed. More forms of back-jumping, based on reasons why conflicts during STPG have occurred (i.e., conflict-directed backtracking), have to be investigated.
9.2. Suggestions for future research

- ATPG for sequential circuits has been identified as very computing time expensive. Therefore, much advantage is to be expected of parallel and distributed processing of the ATPG task. Many levels of parallelism can be explored in ATPG, varying from coarse (parallel circuit) to fine-grain (parallel search) parallelism.

- Automatic selection of STPG strategies and automatic determination of parameter-values which influence the ATPG process, based on circuits characteristics could be of great value. We have proposed strategy selection based on circuit depth and sizes of FFRs. However, there are much more circuit characteristics which can influence the ATPG process. Research of the properties of these circuit characteristics in ATPG may improve the automatic selection of strategies and parameter-values.

- We expect that for very hard faults the use of the IS identification and expansion techniques in the fault-dependent state space will yield significant test generation improvements, similar to what has been gained for the fault-independent circuit.

- Considering SFF selection, the proposed SFF selection and reduction techniques (see Section 8.5) have to be performed automatically, and new, or other existing SFF selection and/or reduction techniques have to be developed/implemented to improve this selection process.

Furthermore, some topics within the field of sequential circuit testing have not, or not completely, been covered:

- Nowadays, in most circuit designs the FFs are scannable, by default. For each test pattern to be generated, the TPG uses the scan property of the FFs. However, it may improve the test-time and defect coverage of the circuit when the TPG only uses the scan property of the FFs when this is necessary; i.e., the TPG only applies (a) scan-shift(s) during test generation for a fault when this is easier/cheaper than continuing sequentially through the FFs. This is called automatic scan-chain selection during sequential circuit TPG. Much research still has to be done in this field.

- In this thesis, an implicit single clock has been assumed. Research has to be performed to extend the system to handle explicit, multiple clocks, which are very common in current circuit designs.

- In this thesis, the detection of the faults has been a sufficient condition for the test set. However, the generation of diagnostic test sets has not been considered, but is an important topic in the field of test generation.
Bibliography


BIBLIOGRAPHY


Appendix A

ISCAS ’85 and ’89 circuits

Table A.1, and Table A.2 provide characteristics of the combinational ISCAS’85 [Brg85] circuits and the sequential ISCAS’89 [Brg89] circuits. All circuits in both benchmark sets are Boolean designs\(^1\). Column \#conn gives the number of signal lines (which can slightly differ from the number in the name for ISCAS’89 circuits). Columns \#PI, \#PO, \#FF, \#gate, and \#FAN give the numbers of PI’s, PO’s, FF’s, logic gates, and fanout points, respectively. Column \#lv gives the number of logic circuit levels. Column \#Ginp gives the average, and the maximum number of inputs (between braces) of a gate. Column \#brn gives the average, and the maximum number of branches (between braces) of a fanout point. The number of FFRs in a circuit equals \#POs + \#FFs + \#FAN. Column \#FFRblk gives the average, and the maximum number of gates (between braces) in an FFR. Column \#FFRinp gives the average number of inputs, and the maximum number of inputs (between the braces) of an FFR. Columns \#un0, \#un1, \#unob give the numbers of signal lines that are uninitializable to 0, uninitializable to 1, and unobservable at POs (including the potentially unobservable ones (see Section 5.1.3, Page 81)), respectively.

\(^1\)or have been made Boolean for the purpose of the benchmark set.

<table>
<thead>
<tr>
<th>Name</th>
<th>#conn</th>
<th>#PI</th>
<th>#PO</th>
<th>#gate</th>
<th>#FAN</th>
<th>#FFRinp</th>
<th>#lv</th>
</tr>
</thead>
<tbody>
<tr>
<td>c432</td>
<td>432</td>
<td>36</td>
<td>7</td>
<td>160</td>
<td>89</td>
<td>2.83(45)</td>
<td>17</td>
</tr>
<tr>
<td>c499</td>
<td>499</td>
<td>41</td>
<td>32</td>
<td>202</td>
<td>59</td>
<td>3.26(16)</td>
<td>11</td>
</tr>
<tr>
<td>c880</td>
<td>880</td>
<td>60</td>
<td>26</td>
<td>383</td>
<td>125</td>
<td>3.29(17)</td>
<td>24</td>
</tr>
<tr>
<td>c1355</td>
<td>1355</td>
<td>41</td>
<td>32</td>
<td>546</td>
<td>259</td>
<td>2.78(16)</td>
<td>24</td>
</tr>
<tr>
<td>c1908</td>
<td>1908</td>
<td>33</td>
<td>25</td>
<td>880</td>
<td>385</td>
<td>2.51(51)</td>
<td>40</td>
</tr>
<tr>
<td>c2670</td>
<td>2670</td>
<td>233</td>
<td>140</td>
<td>1193</td>
<td>454</td>
<td>2.49(72)</td>
<td>32</td>
</tr>
<tr>
<td>c3540</td>
<td>3540</td>
<td>50</td>
<td>22</td>
<td>1669</td>
<td>579</td>
<td>3.11(62)</td>
<td>47</td>
</tr>
<tr>
<td>c5315</td>
<td>5315</td>
<td>178</td>
<td>123</td>
<td>2307</td>
<td>806</td>
<td>3.24(19)</td>
<td>49</td>
</tr>
<tr>
<td>c6288</td>
<td>6288</td>
<td>32</td>
<td>32</td>
<td>2416</td>
<td>1456</td>
<td>2.60(4)</td>
<td>124</td>
</tr>
<tr>
<td>c7552</td>
<td>7552</td>
<td>207</td>
<td>108</td>
<td>3512</td>
<td>1300</td>
<td>2.87(35)</td>
<td>43</td>
</tr>
</tbody>
</table>

Table A.1: ISCAS ’85 circuit characteristics
<table>
<thead>
<tr>
<th>Name</th>
<th>#conn</th>
<th>#Pl</th>
<th>#PO</th>
<th>#FF</th>
<th>#gate</th>
<th>#Ginp</th>
<th>#lv</th>
<th>FAN</th>
<th>#brn</th>
<th>#FFRbkl</th>
<th>#FFRinp</th>
<th>#un0</th>
<th>#un1</th>
<th>#unob</th>
</tr>
</thead>
<tbody>
<tr>
<td>s27</td>
<td>26</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>10</td>
<td>1.8(2)</td>
<td>6</td>
<td>4</td>
<td>2.3(3)</td>
<td>1.3(4)</td>
<td>2.0(5)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s208</td>
<td>208</td>
<td>11</td>
<td>2</td>
<td>8</td>
<td>96</td>
<td>1.7(3)</td>
<td>14</td>
<td>32</td>
<td>2.9(6)</td>
<td>2.3(35)</td>
<td>2.7(27)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s208.1</td>
<td>218</td>
<td>10</td>
<td>1</td>
<td>8</td>
<td>104</td>
<td>1.7(4)</td>
<td>11</td>
<td>28</td>
<td>3.4(10)</td>
<td>2.8(41)</td>
<td>3.1(32)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s298</td>
<td>298</td>
<td>3</td>
<td>6</td>
<td>14</td>
<td>119</td>
<td>2.1(4)</td>
<td>9</td>
<td>34</td>
<td>4.8(13)</td>
<td>2.2(9)</td>
<td>3.3(15)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s344</td>
<td>355</td>
<td>9</td>
<td>11</td>
<td>15</td>
<td>160</td>
<td>1.7(3)</td>
<td>20</td>
<td>40</td>
<td>3.8(8)</td>
<td>2.4(11)</td>
<td>2.7(11)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s349</td>
<td>340</td>
<td>9</td>
<td>11</td>
<td>15</td>
<td>161</td>
<td>1.7(3)</td>
<td>20</td>
<td>41</td>
<td>3.8(8)</td>
<td>2.4(11)</td>
<td>2.7(11)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s382</td>
<td>382</td>
<td>3</td>
<td>6</td>
<td>21</td>
<td>158</td>
<td>1.9(4)</td>
<td>9</td>
<td>49</td>
<td>4.1(21)</td>
<td>2.1(8)</td>
<td>3.0(12)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s386</td>
<td>386</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>159</td>
<td>2.2(4)</td>
<td>11</td>
<td>26</td>
<td>8.2(23)</td>
<td>4.1(21)</td>
<td>5.8(26)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s400</td>
<td>400</td>
<td>3</td>
<td>6</td>
<td>21</td>
<td>162</td>
<td>2.0(4)</td>
<td>9</td>
<td>53</td>
<td>4.0(22)</td>
<td>2.0(8)</td>
<td>3.0(12)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s420</td>
<td>420</td>
<td>19</td>
<td>2</td>
<td>16</td>
<td>196</td>
<td>1.7(4)</td>
<td>28</td>
<td>66</td>
<td>2.9(6)</td>
<td>2.3(67)</td>
<td>2.7(51)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s420.1</td>
<td>458</td>
<td>18</td>
<td>1</td>
<td>16</td>
<td>218</td>
<td>1.8(4)</td>
<td>13</td>
<td>58</td>
<td>3.6(14)</td>
<td>2.9(85)</td>
<td>3.2(66)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s444</td>
<td>444</td>
<td>3</td>
<td>6</td>
<td>21</td>
<td>181</td>
<td>1.9(4)</td>
<td>11</td>
<td>65</td>
<td>3.7(22)</td>
<td>2.0(7)</td>
<td>2.9(12)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s510</td>
<td>510</td>
<td>19</td>
<td>7</td>
<td>6</td>
<td>211</td>
<td>2.0(4)</td>
<td>12</td>
<td>73</td>
<td>3.8(13)</td>
<td>2.5(45)</td>
<td>3.5(51)</td>
<td>440</td>
<td>441</td>
<td>503</td>
</tr>
<tr>
<td>s526</td>
<td>526</td>
<td>3</td>
<td>6</td>
<td>21</td>
<td>193</td>
<td>2.3(4)</td>
<td>9</td>
<td>54</td>
<td>5.7(13)</td>
<td>2.4(11)</td>
<td>4.1(20)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s526n</td>
<td>526</td>
<td>3</td>
<td>6</td>
<td>21</td>
<td>193</td>
<td>2.3(4)</td>
<td>9</td>
<td>54</td>
<td>5.7(13)</td>
<td>2.4(11)</td>
<td>4.1(20)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s641</td>
<td>639</td>
<td>35</td>
<td>24</td>
<td>19</td>
<td>379</td>
<td>1.4(4)</td>
<td>74</td>
<td>57</td>
<td>3.6(12)</td>
<td>3.8(19)</td>
<td>2.6(29)</td>
<td>27</td>
<td>47</td>
<td>39</td>
</tr>
<tr>
<td>s713</td>
<td>713</td>
<td>35</td>
<td>25</td>
<td>19</td>
<td>393</td>
<td>1.5(4)</td>
<td>74</td>
<td>80</td>
<td>3.3(12)</td>
<td>3.2(13)</td>
<td>2.6(28)</td>
<td>31</td>
<td>51</td>
<td>43</td>
</tr>
<tr>
<td>s820</td>
<td>820</td>
<td>18</td>
<td>19</td>
<td>5</td>
<td>289</td>
<td>2.6(4)</td>
<td>10</td>
<td>39</td>
<td>13.0(49)</td>
<td>4.6(53)</td>
<td>8.4(83)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s832</td>
<td>832</td>
<td>18</td>
<td>19</td>
<td>5</td>
<td>287</td>
<td>2.7(4)</td>
<td>10</td>
<td>39</td>
<td>13.4(50)</td>
<td>4.6(52)</td>
<td>8.7(90)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s838</td>
<td>838</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>390</td>
<td>1.4(4)</td>
<td>56</td>
<td>134</td>
<td>2.8(3)</td>
<td>2.3(125)</td>
<td>2.7(99)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s838.1</td>
<td>938</td>
<td>34</td>
<td>1</td>
<td>32</td>
<td>446</td>
<td>1.8(4)</td>
<td>17</td>
<td>118</td>
<td>3.6(22)</td>
<td>3.0(173)</td>
<td>3.3(134)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s953</td>
<td>953</td>
<td>16</td>
<td>23</td>
<td>29</td>
<td>395</td>
<td>1.8(4)</td>
<td>16</td>
<td>158</td>
<td>3.2(13)</td>
<td>1.9(20)</td>
<td>2.7(19)</td>
<td>510</td>
<td>526</td>
<td>773</td>
</tr>
<tr>
<td>s1196</td>
<td>1196</td>
<td>14</td>
<td>14</td>
<td>18</td>
<td>529</td>
<td>1.9(4)</td>
<td>24</td>
<td>155</td>
<td>4.1(17)</td>
<td>2.8(30)</td>
<td>3.6(29)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s1238</td>
<td>1238</td>
<td>14</td>
<td>14</td>
<td>18</td>
<td>508</td>
<td>2.1(4)</td>
<td>22</td>
<td>165</td>
<td>4.2(19)</td>
<td>2.6(24)</td>
<td>3.7(37)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s1423</td>
<td>1423</td>
<td>17</td>
<td>5</td>
<td>6</td>
<td>657</td>
<td>1.8(4)</td>
<td>59</td>
<td>180</td>
<td>3.8(43)</td>
<td>2.5(22)</td>
<td>3.0(20)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s1488</td>
<td>1488</td>
<td>8</td>
<td>19</td>
<td>6</td>
<td>653</td>
<td>2.1(4)</td>
<td>17</td>
<td>76</td>
<td>10.8(55)</td>
<td>6.5(56)</td>
<td>8.3(75)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s1494</td>
<td>1494</td>
<td>8</td>
<td>19</td>
<td>6</td>
<td>647</td>
<td>2.2(4)</td>
<td>17</td>
<td>76</td>
<td>11.0(56)</td>
<td>6.4(56)</td>
<td>8.4(76)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s3578</td>
<td>5295</td>
<td>35</td>
<td>49</td>
<td>179</td>
<td>2779</td>
<td>1.5(4)</td>
<td>25</td>
<td>855</td>
<td>2.7(10)</td>
<td>2.6(146)</td>
<td>2.3(122)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s9234</td>
<td>9234</td>
<td>19</td>
<td>22</td>
<td>228</td>
<td>5597</td>
<td>1.4(4)</td>
<td>58</td>
<td>1013</td>
<td>3.4(32)</td>
<td>4.4(217)</td>
<td>2.9(165)</td>
<td>7070</td>
<td>7075</td>
<td>9031</td>
</tr>
<tr>
<td>s9234.1</td>
<td>9234</td>
<td>36</td>
<td>39</td>
<td>211</td>
<td>5597</td>
<td>1.4(4)</td>
<td>58</td>
<td>1013</td>
<td>3.4(32)</td>
<td>4.4(217)</td>
<td>2.9(165)</td>
<td>26</td>
<td>10</td>
<td>3899</td>
</tr>
</tbody>
</table>

Table A.2: ISCAS '89 circuit characteristics
Appendix B

Industrial circuit designs

Table B.1 provides characteristics of the combinational industrial circuits and the sequential industrial circuits in full-scan mode. All latches and flipflops in these sequential circuits have been replaced by pairs of a PPI and a corresponding PPO. Tables B.2 and B.3 provide characteristics of sequential industrial circuits (in non-scan mode). Table B.4 and Table B.5 provide characteristics of the 80C51 circuit and some of its modules. Most of the circuit design in this appendix have been obtained from Philips Electronics Design & Test. Some of the circuits contain 3-state elements.

Column #conn gives the number of signal lines. Columns #PI, #PO, #FF, #gate, #bus, #sw, and #FAN, give the numbers of PIs (including PPIs), POs (including PPOs), FFs, logic gates, 3-state buses, 3-state switches, and fanout points, respectively. Column #lev gives the number of logic circuit levels. Column #Ginp gives the average, and the maximum number of inputs (between braces) of a gate. Column #brn gives the average, and the maximum number of branches (between braces) of a fanout point. The number of FFRs in a circuit equals #POs + #FFs + #FAN. Column #FFRblk gives the average, and the maximum number of gates (between braces) in an FFR. Column #FFRinp gives the average, and the maximum number of inputs (between braces) of an FFR. Columns #nn0, #nn1, #nnob give the numbers of signal lines that are uninitializable to 0, uninitializable to 1, and unobservable at POs (including the potentially unobservable ones (see Section 5.1.3, Page 81)), respectively.
<table>
<thead>
<tr>
<th>Name</th>
<th>#conn</th>
<th>#PI</th>
<th>#PO</th>
<th>#gate</th>
<th>#bus</th>
<th>#sw</th>
<th>#FAN</th>
<th>#FFRinp</th>
<th>#lev</th>
</tr>
</thead>
<tbody>
<tr>
<td>p833</td>
<td>833</td>
<td>112</td>
<td>81</td>
<td>41</td>
<td>120</td>
<td>160</td>
<td>72</td>
<td>3.35(40)</td>
<td>4</td>
</tr>
<tr>
<td>p2133</td>
<td>2118</td>
<td>206</td>
<td>110</td>
<td>944</td>
<td>0</td>
<td>0</td>
<td>370</td>
<td>2.45(17)</td>
<td>27</td>
</tr>
<tr>
<td>p2183</td>
<td>2183</td>
<td>53</td>
<td>5</td>
<td>711</td>
<td>0</td>
<td>0</td>
<td>276</td>
<td>5.24(39)</td>
<td>47</td>
</tr>
<tr>
<td>p2736</td>
<td>2736</td>
<td>148</td>
<td>154</td>
<td>1252</td>
<td>8</td>
<td>20</td>
<td>344</td>
<td>2.91(32)</td>
<td>23</td>
</tr>
<tr>
<td>p7532</td>
<td>7532</td>
<td>212</td>
<td>220</td>
<td>3254</td>
<td>24</td>
<td>0</td>
<td>863</td>
<td>3.93(56)</td>
<td>40</td>
</tr>
<tr>
<td>p8429</td>
<td>8429</td>
<td>608</td>
<td>661</td>
<td>2630</td>
<td>397</td>
<td>711</td>
<td>1048</td>
<td>2.74(44)</td>
<td>23</td>
</tr>
<tr>
<td>p24694</td>
<td>24694</td>
<td>24178</td>
<td>24162</td>
<td>208</td>
<td>0</td>
<td>92</td>
<td>1.01(14)</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>p26603</td>
<td>26603</td>
<td>1576</td>
<td>1525</td>
<td>7683</td>
<td>583</td>
<td>1202</td>
<td>2943</td>
<td>3.84(264)</td>
<td>86</td>
</tr>
<tr>
<td>p32118</td>
<td>32118</td>
<td>1572</td>
<td>1560</td>
<td>14396</td>
<td>16</td>
<td>34</td>
<td>4387</td>
<td>2.97(244)</td>
<td>53</td>
</tr>
<tr>
<td>p34592</td>
<td>34592</td>
<td>408</td>
<td>651</td>
<td>17685</td>
<td>0</td>
<td>0</td>
<td>4163</td>
<td>3.51(92)</td>
<td>98</td>
</tr>
<tr>
<td>p34595</td>
<td>35325</td>
<td>1471</td>
<td>1438</td>
<td>11204</td>
<td>603</td>
<td>2076</td>
<td>3778</td>
<td>4.11(180)</td>
<td>48</td>
</tr>
<tr>
<td>p63578</td>
<td>63578</td>
<td>849</td>
<td>579</td>
<td>32853</td>
<td>0</td>
<td>0</td>
<td>8851</td>
<td>3.26(103)</td>
<td>60</td>
</tr>
<tr>
<td>p65991</td>
<td>65931</td>
<td>2448</td>
<td>2448</td>
<td>28927</td>
<td>12</td>
<td>0</td>
<td>10016</td>
<td>2.97(76)</td>
<td>81</td>
</tr>
<tr>
<td>p132721</td>
<td>132721</td>
<td>5506</td>
<td>5498</td>
<td>63923</td>
<td>26</td>
<td>94</td>
<td>16395</td>
<td>3.14(87)</td>
<td>50</td>
</tr>
<tr>
<td>p235318</td>
<td>235278</td>
<td>10347</td>
<td>9576</td>
<td>116914</td>
<td>78</td>
<td>24</td>
<td>25858</td>
<td>3.34(335)</td>
<td>82</td>
</tr>
</tbody>
</table>

Table B.1: Characteristics of the combinational and sequential (in full-scan mode) industrial circuits

<table>
<thead>
<tr>
<th>Name</th>
<th>#conn</th>
<th>#PI</th>
<th>#PO</th>
<th>#FF</th>
<th>#gate</th>
<th>#bus</th>
<th>#sw</th>
<th>#Ginp</th>
<th>#lev</th>
</tr>
</thead>
<tbody>
<tr>
<td>p169</td>
<td>169</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>53</td>
<td>8</td>
<td>4</td>
<td>2.02(5)</td>
<td>7</td>
</tr>
<tr>
<td>p314</td>
<td>314</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>139</td>
<td>0</td>
<td>0</td>
<td>2.02(5)</td>
<td>9</td>
</tr>
<tr>
<td>p660</td>
<td>660</td>
<td>6</td>
<td>2</td>
<td>16</td>
<td>209</td>
<td>32</td>
<td>16</td>
<td>2.03(5)</td>
<td>14</td>
</tr>
<tr>
<td>p1514</td>
<td>1514</td>
<td>18</td>
<td>33</td>
<td>55</td>
<td>645</td>
<td>0</td>
<td>0</td>
<td>1.90(4)</td>
<td>49</td>
</tr>
<tr>
<td>p2024</td>
<td>2024</td>
<td>19</td>
<td>16</td>
<td>47</td>
<td>570</td>
<td>38</td>
<td>68</td>
<td>2.69(4)</td>
<td>10</td>
</tr>
<tr>
<td>p2133</td>
<td>2118</td>
<td>144</td>
<td>33</td>
<td>77</td>
<td>944</td>
<td>0</td>
<td>0</td>
<td>1.74(3)</td>
<td>27</td>
</tr>
<tr>
<td>p2756</td>
<td>2756</td>
<td>27</td>
<td>3</td>
<td>36</td>
<td>1207</td>
<td>0</td>
<td>0</td>
<td>1.85(3)</td>
<td>102</td>
</tr>
<tr>
<td>p7273</td>
<td>7273</td>
<td>160</td>
<td>48</td>
<td>186</td>
<td>3341</td>
<td>0</td>
<td>0</td>
<td>1.84(8)</td>
<td>52</td>
</tr>
<tr>
<td>p24938</td>
<td>24938</td>
<td>54</td>
<td>6</td>
<td>347</td>
<td>7149</td>
<td>0</td>
<td>0</td>
<td>3.15(263)</td>
<td>93</td>
</tr>
</tbody>
</table>

Table B.2: Characteristics of the sequential industrial circuits (part A)

<table>
<thead>
<tr>
<th>Name</th>
<th>#FAN</th>
<th>#brn</th>
<th>#FFRblk</th>
<th>#FFRinp</th>
<th>#unk0</th>
<th>#unk1</th>
<th># unkob</th>
</tr>
</thead>
<tbody>
<tr>
<td>p169</td>
<td>28</td>
<td>3.21(8)</td>
<td>1.71(6)</td>
<td>2.74(14)</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>p314</td>
<td>23</td>
<td>7.26(17)</td>
<td>4.21(31)</td>
<td>5.30(30)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>p660</td>
<td>105</td>
<td>3.48(20)</td>
<td>1.85(6)</td>
<td>2.90(14)</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>p1514</td>
<td>201</td>
<td>3.96(36)</td>
<td>2.23(12)</td>
<td>3.01(10)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>p2024</td>
<td>141</td>
<td>9.09(170)</td>
<td>3.31(38)</td>
<td>6.61(94)</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>p2133</td>
<td>370</td>
<td>2.62(13)</td>
<td>1.97(22)</td>
<td>2.45(17)</td>
<td>1859</td>
<td>1876</td>
<td>1974</td>
</tr>
<tr>
<td>p2756</td>
<td>374</td>
<td>3.81(32)</td>
<td>3.07(68)</td>
<td>3.61(63)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>p7273</td>
<td>895</td>
<td>4.02(32)</td>
<td>2.96(38)</td>
<td>3.48(48)</td>
<td>800</td>
<td>1629</td>
<td>1215</td>
</tr>
<tr>
<td>p24938</td>
<td>2054</td>
<td>8.47(259)</td>
<td>2.97(53)</td>
<td>7.39(263)</td>
<td>23404</td>
<td>23863</td>
<td>24787</td>
</tr>
</tbody>
</table>

Table B.3: Characteristics of the sequential industrial circuits (part B)
### Table B.4: Characteristics of the 80C51 circuit and modules (part A)

<table>
<thead>
<tr>
<th>Name</th>
<th>#conn</th>
<th>#PI</th>
<th>#PO</th>
<th>#FF</th>
<th>#gate</th>
<th>#bus</th>
<th>#sw</th>
<th>#Ginp</th>
<th>#lev</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>2331</td>
<td>46</td>
<td>50</td>
<td>39</td>
<td>1093</td>
<td>0</td>
<td>0</td>
<td>1.80(5)</td>
<td>32</td>
</tr>
<tr>
<td>ASF</td>
<td>8412</td>
<td>47</td>
<td>91</td>
<td>314</td>
<td>4013</td>
<td>0</td>
<td>0</td>
<td>1.75(3)</td>
<td>37</td>
</tr>
<tr>
<td>CORE</td>
<td>16423</td>
<td>88</td>
<td>177</td>
<td>468</td>
<td>7821</td>
<td>0</td>
<td>0</td>
<td>1.77(8)</td>
<td>51</td>
</tr>
<tr>
<td>CPU</td>
<td>7904</td>
<td>12</td>
<td>72</td>
<td>184</td>
<td>3728</td>
<td>0</td>
<td>0</td>
<td>1.80(8)</td>
<td>45</td>
</tr>
<tr>
<td>I2C</td>
<td>1993</td>
<td>20</td>
<td>12</td>
<td>65</td>
<td>922</td>
<td>0</td>
<td>0</td>
<td>1.77(3)</td>
<td>36</td>
</tr>
<tr>
<td>INT7L2</td>
<td>3395</td>
<td>37</td>
<td>20</td>
<td>91</td>
<td>1595</td>
<td>0</td>
<td>0</td>
<td>1.77(3)</td>
<td>36</td>
</tr>
<tr>
<td>TIMER2</td>
<td>2235</td>
<td>22</td>
<td>16</td>
<td>52</td>
<td>1098</td>
<td>0</td>
<td>0</td>
<td>1.78(3)</td>
<td>22</td>
</tr>
<tr>
<td>UART</td>
<td>1324</td>
<td>23</td>
<td>14</td>
<td>65</td>
<td>623</td>
<td>0</td>
<td>0</td>
<td>1.71(3)</td>
<td>30</td>
</tr>
<tr>
<td>WDT</td>
<td>583</td>
<td>18</td>
<td>11</td>
<td>29</td>
<td>279</td>
<td>0</td>
<td>0</td>
<td>1.66(3)</td>
<td>33</td>
</tr>
</tbody>
</table>

### Table B.5: Characteristics of the 80C51 circuit and modules (part B)

<table>
<thead>
<tr>
<th>Name</th>
<th>#FAN</th>
<th>#brn</th>
<th>#FFRblk</th>
<th>#FFRinp</th>
<th>#un0</th>
<th>#un1</th>
<th>#unoh</th>
</tr>
</thead>
<tbody>
<tr>
<td>80C51</td>
<td>2937</td>
<td>4.05(43)</td>
<td>3.14(120)</td>
<td>3.37(98)</td>
<td>12308</td>
<td>14217</td>
<td>23803</td>
</tr>
<tr>
<td>ALU</td>
<td>274</td>
<td>4.21(19)</td>
<td>3.01(36)</td>
<td>3.41(32)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ASF</td>
<td>1004</td>
<td>4.02(38)</td>
<td>2.85(120)</td>
<td>3.12(92)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CORE</td>
<td>1964</td>
<td>4.10(43)</td>
<td>3.00(69)</td>
<td>3.30(67)</td>
<td>227</td>
<td>253</td>
<td>298</td>
</tr>
<tr>
<td>CPU</td>
<td>932</td>
<td>4.27(33)</td>
<td>3.14(69)</td>
<td>3.52(67)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>I2C</td>
<td>280</td>
<td>3.52(23)</td>
<td>2.58(42)</td>
<td>3.00(39)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>INT7L2</td>
<td>462</td>
<td>3.62(18)</td>
<td>2.78(26)</td>
<td>3.14(23)</td>
<td>2</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>TIMER2</td>
<td>208</td>
<td>5.11(39)</td>
<td>3.98(120)</td>
<td>4.12(92)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>UART</td>
<td>182</td>
<td>3.37(13)</td>
<td>2.39(15)</td>
<td>2.69(16)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>WDT</td>
<td>79</td>
<td>3.25(14)</td>
<td>2.34(11)</td>
<td>2.55(8)</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Appendix C

DAT: Delft Advanced Test generation tool

The algorithms and techniques in this thesis have been implemented in C++ inside the ATPG system: DAT, the Delft Advanced Test generation system. Details on the implementation can be found in [Kon96a]. So far, the code has been successfully ported to Sun (Sun4, Sparc I/II; SunOS, Solaris), HP (PA-Risc; HP-UX), Silicon graphics (IRIX), DEC, and Linux platforms.

The DAT tool can be used fully stand-alone, and can be operated by means of (scripts or command line input in) a dedicated command language (DCL [vdL96] [Kon96b]). It can read in circuits in the ISCAS formats, 2 internal formats, KISS2 [Kis89], BLIF [Bli89], Verilog (limited), and the (Philips) NDL format (and via AMSAL, also EDIF 1.1.0 and EDIF 2.0.0).

The main tasks which DAT can perform are ATPG (using random and deterministic (extended FAN-algorithm based) test pattern generation (TPG)), fault simulation (FS), and logic simulation (LS), for sequential as well as combinational circuits. These circuits may contain one or more scan-chains, which can be explicitly or implicitly present in the circuits.

Sequential circuit TPG is performed based on the modified combinational (FAN-algorithm based) TPG algorithms, by applying the forward time processing methodology during fault propagation, and the backward time processing methodology during the justification process of requirements. Random TPG, FS and LS can be performed ‘parallel patterns’ (32 patterns at the same time). Compact ATPG by means of extended additional targets TPG is supported as well. Fault model/test methods combinations supported for combinational circuits are: Single Stuck-At Fault (SSAF) logic observation testing, and potential logic observation testing, and Iddq testing of SSAFs and bridging faults. For sequential circuits the Single Stuck-At Fault (SSAF) logic observation testing is supported.

By means of the command language, very complex ATPG schemes (having multiple stages, each using multiple strategies in test generation) can be defined, and ‘aliased’ into a simple user-defined command. Apart from Boolean gates, various 3-state elements such as (regular 3-state, ‘wired’, and ‘pulled’) buses, and various types of switches and drivers, as well as bidirectional terminals (I/O pins) are supported.

Since early 1995, the combinational circuit part of DAT has replaced the heart of
Philips' AMSAL [Hap89] ATPG system. The various interfacing procedures of AMSAL have been recoded in such a way that the external interface to AMSAL remains practically the same as it used to be, such as net-list compilation, pattern conversions, etc. AMSAL is used as the ATPG tool within Philips design and production departments worldwide. AMSAL in its turn, is an important part of the larger test generation system, that also incorporates test generation algorithms for various types of RAM, ROM, etc.
Samenvatting (summary in Dutch)

Automatische test patroon generatie voor synchrone sequentiële schakelingen

Om te kunnen garanderen dat geproduceerde chips correct functioneren is het structureel testen van deze chips op eventuele defecten noodzakelijk. Voor het structureel testen wordt automatisch een testset gegenereerd uitgaande van een foutmodel.

Het onderwerp van dit proefschrift is automatische test patroon generatie (ATPG) voor *synchrone sequentiële digitale schakelingen* op basis van het ‘Single Stuck-at Fault’ (SSAF) foutmodel. Door de zeer hoge complexiteit van ATPG voor sequentiële schakelingen is dit onderwerp nog steeds een onderzoeksgebied. Hoge fout efficiënties kunnen voor veel van deze schakelingen niet of pas na zeer lange ATPG rekentijden behaald worden; dit in tegenstelling tot combinatorische schakelingen, waar hoge fout efficiënties regel zijn. Er bestaan oplossingen om de test generatie complexiteit terug te brengen tot die van vergelijkbare combinatorische schakelingen, maar die hebben als nadeel verminderde prestaties van de chip, een groter chip oppervlak en langere test applicatie tijden. Het doel van dit onderzoek is een ATPG systeem te ontwikkelen voor sequentiële schakelingen dat hogere fout efficiënties bereikt en/of lagere test generatie tijden heeft in vergelijking met eerder gepubliceerd werk.

Na een uitgebreide inleiding van sequentiële schakelingen en ATPG in Hoofdstuk 2 en 3, worden in Hoofdstuk 4 drie problemen in STPG geïdentificeerd welke tot op heden niet opgelost zijn: Het zoeken naar een oplossing in STPG is vaak incompleet en inefficient, bus-conflicten worden soms ten onrechte verondersteld aanwezig te zijn in een schakeling, en illegale toestanden (ITs) zijn vaak over-gespecificeerd. Deze problemen zijn gerelateerd aan het gebruikte signaal model tijdens STPG en aan de identificatie van ITs. Gebaseerd op deze problemen en eerder gepubliceerd werk zijn open vragen en voorstellen opgesteld die in de latere hoofdstukken beantwoord en behandeld worden.

Hoofdstuk 5 beschrijft het ontwikkelde ATPG systeem voor sequentiële schakelingen; het bestaat uit een STPG gebaseerd op het deterministische FAN algoritme, en een aantal nieuwe pre-processen en nieuwe leer technieken ontwikkeld ter ondersteuning van de STPG. De STPG bevat behalve bestaande technieken, een aantal nieuwe technieken zoals fout afhankelijke kosten schattingen ter ondersteuning van het beslissingsproces tijdens STPG, en de ‘State Restrictor’ voor opslag van IT’s, waarmee voorkomen wordt dat bekende IT’s opnieuw gejustificeerd worden tijdens STPG.

Hoofdstuk 6 introduceert een nieuw signaal model: de *Power Set Logica* (PSL). Dit model wordt gevormd door de power-set (d.w.z. de set van alle partiële sets) van de set \( \{0, 1, Z\} \). Met dit model is het bijvoorbeeld mogelijk om uit te drukken dat een signaal
Samenvatting

lijn nog θ of Z kan worden of dat een signaal lijn een onbekende waarde heeft, maar
dat die waarde of θ of Z moet zijn. PSL lost twee van de drie problemen genoemd in
Hoofdstuk 4 op: Het zoek-proces tijdens STPG is gegarandeerd compleet en incorrekte
bus-conflict detectie kan niet voorkomen. Verder garandeert PSL dat alle beslissingen
genomen tijdens STPG deterministisch zijn, in tegenstelling tot de soms willekeurige her-
beslissingen in STPGs gebaseerd op enumeratieve signaal modellen. Door deze garantie
van PSL verbetert de zoek-efficiëntie van de STPGs in grote mate. Ten slotte kan genoemd
worden dat de resolutie van PSL veel hoger is dan bij andere signaal modellen waardoor
de signaal waarden op de lijnen veel preciezer berekend kunnen worden. De experimenten
bewijzen de kracht van PSL ten opzichte van andere signaal modellen: Significant hogere
fout efficiënties (gemiddeld zijn de fout efficiënties voor de schakelingen met 8% geste-
gen) worden bereikt voor zowel industriële als ISCAS’89 benchmarks schakelingen, met
daarbij een reductie van CPU kosten (gemiddeld zijn de CPU kosten met 30% gedaald).

Hoofdstuk 7 beschrijft nieuwe technieken om de tot nu toe bekende IT ruimte (de
ruimte die omvat wordt door de lijst van huidig bekende IT’s) uit te breiden. Deze
technieken hebben als doel over-specificaties in de huidig bekende IT’s te identificeren,
waarmee het derde probleem beschreven in Hoofdstuk 4 wordt opgelost. Gebaseerd op
de bekende IT’s worden nieuwe kandidaat IT’s gevormd, welke na een falende justificatie
door de STPG ook IT worden. Deze nieuwe IT’s omvatten minimaal de IT’s waarmee ze
gevormd werden. Deze omvatte IT’s worden uit de lijst verwijderd, wat geheugen ruimte
bespaard, en tegelijkertijd is de bekende IT ruimte gegroeid. Een nieuw leer proces wordt
beschreven om de IT’s te bepalen voordat STPG begint. Tenslotte wordt een nieuwe data-
structuur beschreven voor de opslag van de IT’s, waardoor veel geheugen bespaard wordt
ten verwerkings snelheid van de IT’s significant verhoogd wordt. De experimenten
tonen zeer hoge fout efficiënties, behaald binnen een redelijke tijd, voor de meeste IS-
CAS’89 schakelingen en industriële schakelingen. Voor 20 van de eerste 25 ISCAS’89
schakelingen is 100% fout efficiëntie behaald. Voor slechts 6 van de 38 schakelingen
is de bereikte fout efficiëntie lager dan 90%. In vergelijking met eerder gepubliceerde
resultaten is dit een grote vooruitgang.

Hoofdstuk 8 beschrijft de resultaten van een case-studie om de effectiviteit en ef-
ficiëntie van het ontwikkelde ATPG systeem te demonstreren. De ‘case’ is de Philips
80C51 micro-controller. ATPG is verricht op een ‘full-scan’ versie, een volledig sequen-
tiële versie en op diverse ‘partial-scan’ versies van die schakeling. De resultaten
tonen de effectiviteit en efficiëntie van de nieuw ontwikkelde STPG aan; hoge fout efficiënties
worden behaald binnen een redelijke test generatie tijd. Verder laten de resulta-
taten zien dat een ‘partial-scan’ versie met ongeveer 50% scan-flipflops een fout dekking
oplevert die vergelijkbaar is met de fout dekking voor de ‘full-scan’ versie.

Tot slot wordt in Hoofdstuk 9 een samenvatting gegeven van het onderzoek, en worden
de belangrijkste bijdragen opgesomd en aanbevelingen gedaan voor toekomstige uitbrei-
dingen en verbeteringen.

Mario Konijnenburg
Curriculum vitae

Mario Konijnenburg was born in Ridderkerk on December 5, 1968. From 1987 to 1992 he studied at the department of Electrical Engineering of the Delft University of Technology, where he graduated in 1992. During his MSc study he performed a literature research on Iddq testing and implemented a test pattern generator for Iddq testing based on the leakage fault model. In 1992 he joined Prof. A.J. van de Goor’s computer architecture group at the faculty of Electrical Engineering of Delft University as an assistant researcher. His main activity was to develop a test generation tool based on the FAN algorithm for three-state circuits. From March 1995 to December 1998, he has been a PhD student of Prof. van de Goor at the computer architecture group, where he performed the research described in this thesis. This research has been funded by the Nederlandse organisatie voor Wetenschappelijk Onderzoek (NWO) and Philips Electronic N.V. A large part of the software, which has been developed over the years 1992 to 1998 during the work at the computer architecture group, is now in use as the Philips’ production ATPG system worldwide. Mario’s (research) interests include all aspects of IC testing, but also complex software problems, such as the (NP-complete) search problems in TPG.
Index

3-state circuit, 15, 56
3-state element, 15
80C51, 172
aborted fault, 29
additional target TPG, 117
application-mode test, 2
asynchronous input, 17
asynchronous reset, 101
asynchronous sequential circuit, 4
ATE, 1
ATPG stage, 36
ATPG system, 34
automatic test equipment, 1
BACK, 50
back-jumping, 101
backtrace, 41, 44
backtrace, counters, 94
backtrack mechanism, 131
backtracking, 36
backward global implication, 113
backward justification phase, 85
basic set, 126
basic value, 126
BCD-counter, 60
bistable element, 17
blocked path, 93
Boolean circuit, 14
BPI, 113
branch-and-bound TPG, 39
bridging fault, 26
bus, 16
bus-conflict, 9, 15, 45
bus-conflict detection, 56
bus-padding, 45, 100, 134
bus-patch, 25
can-be-Z, 76
cf_limit, 157
check-illegal-states, 155
CIS, 155
clock, 4
clock pulse, 4
CO, 41
combinational block, 14
combinational circuit, 3
common_fraction_array, 157
complete fault-cone, 89
complete search, 57
complexity of STPG, 4
compressed vector, 152
conflict-free condition, 27
connection, 14
CONSEQUENT, 64
constructive dilemma, 111
Contest, 49
contrapolation, 111
controllability, 7, 76
controllability to Z, 79
controllability, fault dependent, 88
controllability, merged, 78
CRIS, 49
CS, 14
CTP, 49
CTP approach, 51
CTPG, 4
current state, 4, 14
cvector, 152
D latch, 17
D-algorithm, 36
D-flipflop, 18
D-front, 40, 45
D-front update, 94
DAT, 9, 44, 215
DD, 45
DD determination, 91
decision refinement, 132, 137
decision stack, 36, 39
decision tree, 39
decision, three-valued, 44, 94
decision-making process, 57, 135
deep circuit, 103, 105
default value, 128
defect, 2
delay fault, 27
dependency graph, 183
dependency matrix, 183
design for testability, 7
detectable fault, 29
detection condition, combinational, 27
detection condition, sequential, 28
d ETF, 7
differentiating sequence, 28
direct implication, 110
discrepant value, 40
dominator, 40
DPM level, 1
drivability, 88
DTPG, 35
DUST, 51, 69
dynamic dominators, 45

EBT, 50
effective GIS, 144, 155
effective illegal state, 149
effective legal state, 149
EIS, 155
element, 14
Essential, 50
essential value, 134, 136
expand_illegal_state, 154

FAN algorithm, 36, 39
FanO, 42
fanout, reconvergent, 36
fanout-free region, 105
FASTEST, 49
fault, 2
fault activation, 4, 35
fault activation phase, 84
fault activation, combinational, 27
fault activation, sequential, 28
fault collapsing, 34
fault coverage, 5
fault detection condition, 27
fault dropping, 37
fault efficiency, 5
fault model, 26
fault observation, 27
fault observation, combinational, 27
fault observation, sequential, 28
fault propagation, 4, 35, 91
fault propagation phase, 85
fault simulation, 37, 116
fault, aborted, 29
fault, undetected, 29
fault, detectable, 29
fault, equivalent, 34
fault, logical, 26
fault, parametric, 27
fault, potential detectable, 29
fault, redundant, 30
fault, sequential redundant, 30
fault, undetectable, 29
fault, untestable, 29
fault-cone, 88
fault-cone, complete, 89
fault-cone, local, 89
fault-dependent illegal state list, 97
fault-effect, 4
fault-free circuit, 26
fault-sensitive line, 91
fault-site, 40
faulty circuit, 26
feedback loop, 3
FF, 4
FF group, 184
FF-group, 151, 159
FFR, 105
FGL, 113
FILL, 66
finite state machine, 19
FISL, 97
INDEX

fixed value, 111, 127
fixed vector, 25
flipflop, 4, 18
flipflop state, 18
flipflop, positive edge triggered, 19
flipflop, pseudo, 24
FO, 41
FOGBUSTER, 52, 69
forward global implication, 113
forward time processing, 49
FS, 37, 116
FSM, 19
FTP, 49
FUN, 81
functional tests, 2
gate network, 14
gated D latch, 17
gated latch, 17
GIS, 60
GIS expansion, 145
GIS identification, 144, 145
GIS learning, 146
GIS list, 98
GIS list, irredundant, 144
GIS merging, 144
GIS minimization, 65
GIS optimization, 144
GIS, common fraction, 155
GIS, effective, 144, 155
GIS, over-specified, 153
GIS, prime, 144
global illegal state list, 98
global implication, 110
headline, 42
high-impedance, 15
hit-rate value, 148
HITEC, 52, 64–66
HITEST, 50
HO, 42
hyper-cube, 144
hyper-plane, 145
Iddq fault, 27
ignore PPO reached, 106
illegal state, 5, 22, 60
implication, exhaustive, 40
incomplete search, 57
indirect implication, 110
initial state, 21, 28
initialization sequence, 21
input sequence, 14, 24
input stimulus, 14
input vector, 14
internal restriction, 95
invalid state, 22
IO, 41
irredundant GIS list, 144
IS, 5, 60
IS identification, 144
IS merging, 144
IS optimization, 144
IS, common fraction, 155
IS, over-specified, 153
IS, temporary, 99
iterative array model, 24
justification learning, 108
Justifier, 145
known GIS space, 61
latch, 4, 17
learning process, 38
legal state, 5, 22, 95, 147
legal state cache, 148
line, 14
line justification problem, 35
local fault-cone, 89
local implication, 110
logic array model, 24
logic block, 3
logic observation, 27
LS cache, 148
make_new_objs, 86
mandatory assignment, 35
Mealy machine, 19
memory block, 3
MinVal, 157
Moore machine, 19
INDEX

MOSAIC, 52, 65, 70
multiple backtrace, 41
multiple fault-effect, 6

near full-scan, 8
next state, 14
next-state function, 19
normal-mode, 175
NS, 14

objective, 41
objective, current, 41
objective, fanout, 42
objective, final, 41
objective, headline, 42
objective, initial, 41
objective, three-valued, 44
observability, 7, 76, 79
observation time, multiple, 47
observation time, single, 47
OISL, 98
optimal STPG selection, 104
oscillation, 18
OUSL, 98
output function, 19
output response, 14
output sequence, 14, 24
output vector, 14
over-specified IS, 153
over-specified state, 58, 61

padding, random, 37, 45
parametric tests, 2
parity signal line, 91
path-searching algorithm, 36
PGIS, 144, 153
PI, 14
PI space, 61
PO, 14
PODEM algorithm, 36
post-process, 38
POT..ISL, 155
potential detectable fault, 29
potential GIS, 144
power-set, 126
power-set signal model, 126

power-up state, 21
PPI, 14
PPI state, 96
PPO, 14
PPO state, 96
PPO illegal state list, 98
PPO used state list, 98
pre-process, 34
pre-processes, 76
primary input, 14
primary output, 14
prime GIS, 144
primitive element, 14
propagation learning, 109
pseudo primary inputs, 14
pseudo primary outputs, 14
PSL, 126
PSL, associative, 130

random logic, 14
random padding, 37, 45, 131
recursive learning, 108
redundant fault, 30
redundant learned implication, 112
relative timeframe number, 113
reset input, 19
reset state, 21
restriction, internal, 16, 95
restricitor, 25
reverse fault simulation, 38
RT, 113
RTP, 49

S-R latch, 17
SA-0, 26
SA-1, 26
SAF, 26
scan design, 7
scan-chain, 7
scan-chain, explicit, 173
scan-chain, implicit, 173
scan-FF, 7
SCOAP, 34
search space, 5, 61
search space, STPG, 95
<table>
<thead>
<tr>
<th>Index</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential circuit, 4</td>
<td>4</td>
</tr>
<tr>
<td>sequential depth, 6</td>
<td>6</td>
</tr>
<tr>
<td>sequential level, 6</td>
<td>6</td>
</tr>
<tr>
<td>sequential redundant fault, 30</td>
<td>30</td>
</tr>
<tr>
<td>set input, 19</td>
<td>19</td>
</tr>
<tr>
<td>set-reset latch, 17</td>
<td>17</td>
</tr>
<tr>
<td>SFF, 7</td>
<td>7</td>
</tr>
<tr>
<td>signal line, 14</td>
<td>14</td>
</tr>
<tr>
<td>signal line, parity, 91</td>
<td>91</td>
</tr>
<tr>
<td>signal model, 61</td>
<td>61</td>
</tr>
<tr>
<td>signal model, 25V, 44, 63</td>
<td>25V, 44, 63</td>
</tr>
<tr>
<td>signal model, 3V, 15, 61</td>
<td>3V, 15, 61</td>
</tr>
<tr>
<td>signal model, SUV, 16, 24, 61</td>
<td>SUV, 16, 24, 61</td>
</tr>
<tr>
<td>signal model, 5V, 62</td>
<td>5V, 62</td>
</tr>
<tr>
<td>signal model, 9V, 62</td>
<td>9V, 62</td>
</tr>
<tr>
<td>signal model, power-set, 126</td>
<td>126</td>
</tr>
<tr>
<td>signal model, resolution, 24, 62</td>
<td>24, 62</td>
</tr>
<tr>
<td>signal model, symbolic, 135</td>
<td>135</td>
</tr>
<tr>
<td>signal value, 14</td>
<td>14</td>
</tr>
<tr>
<td>SimTPG, 37</td>
<td>37</td>
</tr>
<tr>
<td>single backtrace, 42, 105</td>
<td>42, 105</td>
</tr>
<tr>
<td>single stuck-at fault, 2</td>
<td>2</td>
</tr>
<tr>
<td>single stuck-at fault model, 26</td>
<td>26</td>
</tr>
<tr>
<td>SR, 96</td>
<td>96</td>
</tr>
<tr>
<td>Stallion, 52</td>
<td>52</td>
</tr>
<tr>
<td>state, 4</td>
<td>4</td>
</tr>
<tr>
<td>state initialization, 21</td>
<td>21</td>
</tr>
<tr>
<td>state over-specification, 58</td>
<td>58</td>
</tr>
<tr>
<td>state restrictor, 96</td>
<td>96</td>
</tr>
<tr>
<td>state sequence, 24</td>
<td>24</td>
</tr>
<tr>
<td>state space, 61</td>
<td>61</td>
</tr>
<tr>
<td>state transition diagram, 20</td>
<td>20</td>
</tr>
<tr>
<td>state transition graph, 20</td>
<td>20</td>
</tr>
<tr>
<td>state transition table, 20</td>
<td>20</td>
</tr>
<tr>
<td>state, unjustifiable, 22</td>
<td>22</td>
</tr>
<tr>
<td>state, unreachable, 22</td>
<td>22</td>
</tr>
<tr>
<td>state-encoding, 5, 60</td>
<td>5, 60</td>
</tr>
<tr>
<td>static learning, 109</td>
<td>109</td>
</tr>
<tr>
<td>static learning, line justification, 115</td>
<td>115</td>
</tr>
<tr>
<td>static learning, logic implication, 110</td>
<td>110</td>
</tr>
<tr>
<td>STD, 20</td>
<td>20</td>
</tr>
<tr>
<td>STEED, 50</td>
<td>50</td>
</tr>
<tr>
<td>STG, 20</td>
<td>20</td>
</tr>
<tr>
<td>STG1, 50</td>
<td>50</td>
</tr>
<tr>
<td>STPG, 4</td>
<td>4</td>
</tr>
<tr>
<td>STPG strategy, 103</td>
<td>103</td>
</tr>
<tr>
<td>STPG, 3 phases, 84</td>
<td>84</td>
</tr>
<tr>
<td>structural tests, 2</td>
<td>2</td>
</tr>
<tr>
<td>STT, 20</td>
<td>20</td>
</tr>
<tr>
<td>stuck-at fault, 2</td>
<td>2</td>
</tr>
<tr>
<td>stuck-at fault model, 26</td>
<td>26</td>
</tr>
<tr>
<td>stuck-at fault observation, 27</td>
<td>27</td>
</tr>
<tr>
<td>stuck-at-0 fault, 26</td>
<td>26</td>
</tr>
<tr>
<td>stuck-at-1 fault, 26</td>
<td>26</td>
</tr>
<tr>
<td>sv, 14</td>
<td>14</td>
</tr>
<tr>
<td>switch, 16</td>
<td>16</td>
</tr>
<tr>
<td>switch-padding, 131</td>
<td>131</td>
</tr>
<tr>
<td>symbolic signal model, 135</td>
<td>135</td>
</tr>
<tr>
<td>synchronizing sequence, 21</td>
<td>21</td>
</tr>
<tr>
<td>synchronous input, 17</td>
<td>17</td>
</tr>
<tr>
<td>synchronous sequential circuit, 4</td>
<td>4</td>
</tr>
<tr>
<td>targeted D-front, 67</td>
<td>67</td>
</tr>
<tr>
<td>targeted D-front selection, 105</td>
<td>105</td>
</tr>
<tr>
<td>technique T1, 154</td>
<td>154</td>
</tr>
<tr>
<td>technique T2, 155</td>
<td>155</td>
</tr>
<tr>
<td>technique T3, 156</td>
<td>156</td>
</tr>
<tr>
<td>technique T4, 158</td>
<td>158</td>
</tr>
<tr>
<td>technique T5, 159</td>
<td>159</td>
</tr>
<tr>
<td>technique T6, 160</td>
<td>160</td>
</tr>
<tr>
<td>temporary illegal state, 99</td>
<td>99</td>
</tr>
<tr>
<td>test point, 7</td>
<td>7</td>
</tr>
<tr>
<td>test sequence, 4, 28</td>
<td>4, 28</td>
</tr>
<tr>
<td>test set, 3</td>
<td>3</td>
</tr>
<tr>
<td>test set compaction, 38</td>
<td>38</td>
</tr>
<tr>
<td>test strategy, 103</td>
<td>103</td>
</tr>
<tr>
<td>test vector, 3</td>
<td>3</td>
</tr>
<tr>
<td>test-clock, 173</td>
<td>173</td>
</tr>
<tr>
<td>test-mode, 175</td>
<td>175</td>
</tr>
<tr>
<td>testability, 6</td>
<td>6</td>
</tr>
<tr>
<td>testability signal line, 180</td>
<td>180</td>
</tr>
<tr>
<td>three-state circuit, 15</td>
<td>15</td>
</tr>
<tr>
<td>three-state element, 15</td>
<td>15</td>
</tr>
<tr>
<td>three-valued decision, 94</td>
<td>94</td>
</tr>
<tr>
<td>time processing direction, 49</td>
<td>49</td>
</tr>
<tr>
<td>timeframe, 4, 24</td>
<td>4, 24</td>
</tr>
<tr>
<td>topological path, 50</td>
<td>50</td>
</tr>
<tr>
<td>TPG strategy, 103</td>
<td>103</td>
</tr>
<tr>
<td>TPG, deterministic, 35</td>
<td>35</td>
</tr>
</tbody>
</table>
TPG, simulation-based, 37
transitively law, 110
transparency, 18

U, 15, 25, 44, 57
U and PSL, 129
UFI, 80
undetectable fault, 29
undetected fault, 29
uninitializable line, 81
unique sensitization, 40, 45
unjustified line, 40
unknown value, 15
unobservable line, 81
unobservable, potentially, 83
unspecified value, 15
untestable fault, 29
untestable fault identification, 80
US, 45
used state list, 96
useless learned implication, 112
USL, 96

valid state, 22
value, 14
value resolution, 129

wide circuit, 103
window of timeframes, 67

X, 15
X and PSL, 129
X-path check, 41

Z, 15
Z-discrepant, 45