PhD Thesis:

Constraint-Based Architectural Test Pattern Generation

Balázs Sallay
Department of Measurement and Information Systems
Budapest University of Technology and Economics
H-1521 Budapest, Műegyetem rkp. 9, Hungary

April 8, 2000

Advisor:

Associate Prof. András Pataricza
Department of Measurement and Information Systems
Budapest University of Technology and Economics

Contents

1 Introduction **1**
1.1 Cost factors in testing 1
1.2 Problem statement and contribution 2
1.2.1 Test computation goals 3
1.2.2 Contribution of BudaTest 5
1.2.3 Test execution goals 5
1.2.4 Contribution in WST 7
1.3 Environment and terminology 8
1.4 Outline of the thesis 10

2 Current trends in digital design **12**
2.1 CAD tool features 12
2.2 Digital design flow 13
2.2.1 Synthesizable behavioural VHDL 14
2.2.2 High-level synthesis 14
2.2.3 Architectural VHDL 16
2.2.4 Low-level synthesis 16
2.2.5 Gate-level descriptions 17

3 Previous ATPG approaches **19**
3.1 Fault models 19
3.2 Gate-level ATPG 19
3.2.1 Gate-level fault model 20
3.2.2 Gate-level ATPG algorithms 20
3.2.3 Implication 23
3.2.4 Sequential extension 23
3.2.5 CONTEST 24
3.2.6 Algorithm evaluation 25
3.3 High-level ATPG approaches 25
3.3.1 Architectural approaches 25
3.3.2 Hierarchical testing 27
3.3.3 Behavioural approaches 29

3.4 BudaTest objectives 31

4 Constraint-based modelling 33

4.1 CSP definition 33
4.2 CSP solution 35
4.2.1 Backtracking 35
4.2.2 The decision tree 36
4.3 Existing CSP solving methods 37
4.3.1 CSP preprocessing techniques 38
4.3.2 Forward schemes 38
4.3.3 Backward schemes 39
4.4 Constraint-based ATPG modelling in CONTEST 40
4.4.1 ATPG problem representation in CONTEST 41
4.4.2 CSP solving in CONTEST 42

5 ATPG modelling in BudaTest 43

5.1 The constraint network 43
5.1.1 Variable representation 43
5.1.2 Constraint representation 44
5.1.3 CSP-based ATPG problem formulation 44
5.2 Advantages of the constraint technique 46
5.3 Hierarchical support 48
5.3.1 Hierarchical options in BudaTest 48
5.3.2 Problems in CTDF expansion 49
5.4 Control-dominated and sequential circuits 50
5.4.1 Control-dominated circuits 50
5.4.2 Sequential modelling 50
5.5 Array selection 51
5.6 Fault model 52

6 CSP solution 55

6.1 Backtracking in BudaTest 55
6.2 Forward techniques 56
6.2.1 Implication 56
6.2.2 Interval, masked and set logic 58
6.3 Backjumping 60
6.4 Type-uninterpreted search 61
6.4.1 Node classification 61
6.4.2 Colouring goals 62
6.4.3 Token semantics 63
6.4.4 Constraints in the colouring domain 65
6.4.5 Correctness of the colouring search 66

6.4.6 A colouring example 67
6.4.7 The handling of fan-outs, indexing and slicing 68
6.5 Result evaluation 70
6.5.1 Evaluation of implication-related techniques 71
6.5.2 Backjumping performance 72
6.5.3 Evaluation of the type-uninterpreted search technique 73

7 Manufacturing test on the wafer **76**
7.1 Diagnosis terms and wafer-scale testing 77
7.2 Evaluation of the syndrome 78
7.3 Impact of comparator faults 79
7.4 Pre-diagnosis comparator test session 80
7.5 Fault-tolerant result observation 84
7.6 Fault tolerant comparison model 87
7.7 Wafer implementation 88

A Benchmark circuits **A-1**
A.1 Gate-level circuits A-1
A.1.1 Adder family A-1
A.1.2 ISCAS'85 family A-2
A.2 High-level benchmarks A-2
A.2.1 Combinational multiplier A-3
A.2.2 Greatest common divisor A-3
A.2.3 Bubble sort A-5

B The BudaTest program **B-1**

List of Figures

1.1 In-circuit and functional testing of a board 9

2.1 Digital design flow 14
2.2 Architectural style 16

3.1 Active s-a-1 and wired-or short fault 20
3.2 General sequential circuit 24
3.3 Iterative combinational model of a sequential circuit 24
3.4 Use of high and low-level models 28
3.5 Recursive application of the hierarchical principle 28
3.6 Different architectures of the same behavioural source 30
3.7 Comparison of ATPG entry points 31

4.1 Constraint network hypergraph representation 34
4.2 Exemplary decision tree 37

5.1 ATPG problem represented as a constraint network 46
5.2 Symmetric constraints 47
5.3 Network flattening 48
5.4 Representation of stuck-at faults of wide signals 53

6.1 Phases of Sziray's node classification 62
6.2 Large proportion of duplicated variables 62
6.3 The colouring and the typed domains 63
6.4 Two phases of CSP solving 64
6.5 GCD in 4 frames, all variables in two instances 68
6.6 GCD in 4 frames, some variables eliminated by node classification 68
6.7 GCD in 4 frames, many variables eliminated by colouring 69
6.8 The effect of UNKNOWN (input) 69
6.9 Auxiliary information used in D-propagation 70

7.1 Faulty chips 79
7.2 Comparator slice test session 81
7.3 A simple 1-bit comparator 82

7.4 A simple n-bit comparator 83
7.5 Syndrome collection circuitry 85
7.6 Complete test 86
7.7 Syndrome collection at the gate level 86
7.8 Non-identical chips on the wafer 89

A.1 Full adder A-1
A.2 n-bit adder A-1
A.3 4-bit combinational multiplier A-3
A.4 Greatest common divisor A-4
A.5 Bubble sort A-5

B.1 BudaTest block diagram B-2

List of Tables

1.1 Comparison of different ATPG approaches 5
1.2 Advantages (+) and drawbacks (–) of WST methods 7
1.3 Various testing aspects 10

2.1 Features of representation levels 18

3.1 Meaning of composite values in the D algorithm 21
3.2 D propagation table of an OR gate 21

4.1 CONTEST results 42

5.1 Effect of limited CTDF size on the fault coverage 50
5.2 A constraint representing a wired-and short fault 53

6.1 Intersection results 60
6.2 2-to-1 multiplexer colour set 65
6.3 Multiplier colour set 66
6.4 Implication performance for different techniques 72
6.5 Backjumping performance 73
6.6 Effect of node classification and colouring 74

7.1 Invalidation rule in the PMC model 77
7.2 Violations of the Chwa-Hakimi model 80
7.3 Impact of the behaviour of B 81
7.4 Single-bit comparator faults and detecting patterns 82
7.5 Test patterns for n slice 83
7.6 Gate level fault coverage of the test 87
7.7 Refined comparison model 88

A.1 The adder family A-2
A.2 The ISCAS'85 family A-2
A.3 The multiplier family A-3
A.4 The GCD family A-5
A.5 The bubble sort family A-6

Chapter 1

Introduction

The appropriate testing of manufactured components is an essential requirement in the electronics industry, especially in applications requiring highly reliable devices. Since an electronic device can be damaged during manufacturing, assembly, or normal use due to physical faults, the detection of faults is crucial in any phase of the life cycle of the device. In special applications the erroneous behaviour of a faulty component may sometimes cause inestimable damage in the system. Even if the damage is moderate, the early detection of faulty components should always have an important priority.

1.1 Cost factors in testing

It is evident that testing must be carried out for a circuit after its production or occasionally during its lifetime, even if testing is costly.

We will examine the cost and the quality of the test by identifying some important cost factors, mainly from the point of view of *test patterns*. Note that other cost factors exist, e.g. those related to the applied test technology. However, the cost components listed below are important in every technology.

The prerequisite of every testing method is a proper set of test patterns. The selection of the test patterns has a crucial impact on the test quality, and therefore on the testing cost as well. We will place emphasis on the following aspects:

- **test development time:** The time required to generate the test patterns. The test patterns need to be developed only once for a given circuit, therefore the relative cost of test development depends on the testing volume.

- **test execution costs:** These cost factors apply every time an instance of the given circuit is tested.

  - **fault coverage:** This feature shows what proportion of physical faults from a given fault model is detected by the test sequence. The cost of faults not covered by the test includes the damage caused by the possibly faulty circuit,

or the cost required for testing the faulty component at a higher structural level. It is a common rule in the testing industry that the detection of a fault in a component becomes an order of magnitude more expensive when the component is built in a larger system than when it is tested as a stand-alone circuit.

The fault coverage is highly related to the test development time. Obviously, the higher fault coverage is required, the more time is spent with the more accurate test computations.

- **test size:** The space required to store the test patterns.
- **test execution time:** The time required to apply the patterns. This feature is often proportional to the test size, but can also significantly differ, e.g. when test patterns are generated by a certain algorithm (or randomly).

Placing emphasis on different aspects leads to different test strategies. The applied strategy depends on the testing environment and objectives. For example, random test patterns involve negligible test computation time and storage space. However, the fault coverage may remain severely limited (especially in case of sequential circuits), while the test application time can be long due to redundant vectors that actually do not detect any fault. In large systems fault coverage is a much more important issue than little storage space, and in highly dependable systems it becomes particularly crucial. In order to obtain a high fault coverage a *test pattern generator* (TPG) algorithm is required whose task is to find test patterns for given faults with a guaranteed lower bound on the fault coverage. Certainly, deterministic TPG entails that test computation time increases significantly. It should be noted that test patterns are computed only once for a circuit, therefore a high-volume production can decrease relative test computation time.

1.2 Problem statement and contribution

The present thesis aims at the reduction of the costs that correspond to the test quality aspects described above. The following cost factors are addressed:

1. **test computation time and fault coverage:** I have developed a high-level circuit modelling style and automatic TPG (ATPG) which can efficiently handle high-level digital circuits appearing in the engineering practice. The method provides higher fault coverage under the same time constraints, or, equivalently, can reach a similar fault coverage in a shorter time than traditional gate-level ATPG algorithms. In parallel with this performance improvement, the proposed ATPG method provides much better interface to existing high-level computer-aided design (CAD) methodologies and tools.

2. **test execution time:** I propose techniques that improve the applicability of an existing *wafer-scale testing* (WST) idea. In contrast to traditional manufacturing testing which performs chip tests one by one, diagnosis-based WST allows for their parallel execution. This method is, however, very susceptible to the physical faults of certain additional circuitry which is assumed to be fault-free. The techniques proposed in this work provide fault-tolerant features for the diagnosis-based WST method.

1.2.1 Test computation goals

Logic design and CAD systems have undergone a rapid development in the past ten years. Not only the designed circuits became larger by orders of magnitude (in terms of transistor number), but the abstraction level of circuit descriptions was pushed increasingly higher thanks to the sophisticated CAD tools. This improvement has made traditional (gate-level) ATPG algorithms obsolete and out-of-date. The main problems with these algorithms are the following:

- They suffer very much from the exponential nature of ATPG, an NP-complete problem, as proved by Fujiwara [1]. Gate-level ATPG can be in practice applied to combinational circuits consisting of at most a few thousand gates, or to sequential circuits of a much smaller size.

- They are rather awkwardly interfaced with CAD systems. They require a gate-level circuit description, which in many cases does not appear at all in the design flow, since the design target library is not necessarily a gate library.

On the other hand, an unquestionable advantage of the gate-level tools is their accuracy in the area of fault modelling.

It is commonly accepted in the TPG community that the gate-level efficiency problems can be significantly reduced by raising the abstraction level of TPG. In addition to the advantages of CAD-conform circuit models, another gain expected from high-level ATPG tools is faster operation. The high-level ATPG algorithms can exploit that structured high-level descriptions are not simply a large set of unstructured bit signals and gates. Since many bits of signals of wide word lengths can be handled together and large clusters of gates can be regarded as a single component, the ATPG steps can be performed in a significantly smaller problem space. In addition, as control and data signals are explicitly separated, data and fault propagation features of the circuit can be much better revealed.

However, there is no general agreement on the new ATPG entry level. Many high-level approaches exist that use the so-called *behavioural* description level as the basis for TPG. Unfortunately, fault modelling in these approaches is ambiguous or even inappropriate for certain physical fault classes.

I have chosen the *architectural* circuit description level as the ATPG entry point. Architectural descriptions are structural descriptions with abstract data and high-level components. TPG at this level is as efficient as at the behavioural level due to the compactness of the model, and fault modelling is as accurate as in gate-level tools. It will be shown that component interconnection faults can be directly modelled, while the faults internal to components can be handled by *hierarchical modelling*.

In spite of the variety of existing approaches at this level, there is a lack of methods that could efficiently handle data-dominated high-level circuits with intensive data manipulations. Some of the existing approaches are limited to special circuit types (microprocessors or logic controllers), while others have difficulties with handling intensive data dependencies.

BudaTest, the architectural ATPG method and tool proposed in this work, is intended to meet the need for a general approach that is capable of handling circuits with wide data signals and intensive data manipulation.

BudaTest uses constraint-based circuit and fault modelling. The approach owes much to CONTEST, a constraint-based gate-level ATPG tool proposed by Tilly [25]. He has proven the applicability and the efficiency of the constraint mechanism in gate-level TPG [26].

BudaTest focuses on the following questions:

- What modelling problems arise when the constraint based modelling is applied to typical architectural circuits? How to represent wide domains and high-level component descriptions? What are the other new features in architectural representations, and how to model them?

  Chapter 5 deals with this problems. I identify the modelling aspects where gate-level (enumeration-based) methods cannot be applied, and propose new representation techniques for them. I also identify architectural features whose satisfactory treatment by the high-level constraint model is a new feature of the methodology.

- How can be exploited the high-level information that is present in architectural descriptions? What acceleration techniques do they allow?

  Chapter 6 discusses these issues. The exponential increase in the state space caused by the wide word length is moderated by a type-uninterpreted, token-based search technique. This technique also enables the tool to handle moderately sequential and control-dominated circuits.

  BudaTest is not fine-tuned yet as long as heuristic decision control is regarded. I will show, however, that the application of non-heuristic methods that extensively exploit the high-level characteristics of the architecturally described circuit brings such improvement in terms of fault coverage and time demand that makes it a competent high-level tool.

	gate-level tools	CONTEST	BudaTest
circuit level	gate	gate	architectural
signal type	bit	bit	bit, integer, vector
component set	fix	extendable	open
component representation	table	DBCN	rule-based
fault model	fix (s-a, short)	fix (s-a)	library-based incl. s-a and short
algorithm	fix	constraint solving (customisable)	constraint solving (customisable)

Table 1.1: Comparison of different ATPG approaches

1.2.2 Contribution of BudaTest

BudaTest is an architectural TPG method and tool applicable for highly data-dependent and moderately control-dependent circuits. Its operation is based on the same constraint-based principles as those in CONTEST, but the following features are present only in BudaTest:

- Abstract-typed signals are supported.
- The rule-based high-level representation of components is solved. The component library is open. The extension of the constraint library requires only the programming of a single function in a high-level language (C++).
- The accurate modelling of low-level physical faults in the high-level circuit model is solved by defining single-bit interconnection faults and using hierarchical modelling. The coverage of the gate-level *stuck-at fault model* is proven.
- The modelling problems of using *partially wired signals* (half-words, indexed bits etc.) are solved.
- The high-level data and discrepancy propagation information present in the architectural description is heavily exploited.

Table 1.1 summarises the differences between general gate-level tools, CONTEST, and BudaTest.

1.2.3 Test execution goals

Chapter 7 deals with test execution issues in the environment of *wafer-scale testing* (WST). WST takes place after the production of *integrated circuits* (ICs) and before the

packaging of good chips. Thus WST is a specific test execution environment, but is also the most typical one with a very high testing volume.

A recent idea in the post-manufacturing test of ICs is to replace traditional *automatic test equipment* (ATE) based testing with a concurrently executable comparison-based test of the chips on the wafer. The comparators are wired to the IC outputs and compare the IC responses for identical inputs. The comparison outcomes are evaluated by a subsequent diagnosis algorithm. The diagnosis algorithm regards the entire wafer as a single system and the ICs as the components of this system. Since the *resolution* of the diagnosis is exactly the chip level, the system diagnosis consists in identifying the individual chips as fault-free or faulty using the downloaded comparison results (the *diagnostic syndrome*).

The following list and Table 1.2 show the advantages, drawbacks and problems of the diagnosis-based testing technique compared to the traditional testing method.

- The ATE performs testing by positioning the ATE pins on the wafer chip test points and executing the test chip by chip. Diagnosis-based testing can be performed in parallel. This is an important advantage of diagnosis-based testing, especially when the chip test is long or the number of chips is large.

- At-speed testing is frequently difficult with the ATE, because the disturbance caused by the present ATE may be important when the chips operate at high frequencies. Diagnosis-based testing has no such problems.

- The hardware cost in ATE-based testing is represented by the usually extremely expensive ATE. The ATE machinery must be very accurate in the timing of driven and observed pins and in the physical positioning.

- Diagnosis-based testing needs no reference responses while ATE-based testing does.

- The cost of diagnosis-based testing consists in the area overhead caused by the dedicated circuitry required to execute the comparisons. This circuitry is present in every produced chip. It includes the comparators, comparison collectors, and the wires implementing pattern distribution and result collection.

- The test results of the ICs tested with an ATE do not depend on factors other than the coverage of the executed IC test. In diagnosis-based testing, they depend on the number and distribution of faulty chips as well. Since the classification of ICs depends on the comparisons, incorrectly diagnosed chips may entail that other chips become also wrongly diagnosed. Moreover, the correctness of the entire diagnosis algorithm depends on the validity of the applied *diagnostic invalidation rule*, which prescribes what outcomes are possible between components of fault-free or faulty states.

	ATE-based		diagnosis-based	
execution time	–	sequential	+	parallel
ATE required	–	yes	+	no
reference required	–	yes	+	no
speed problems	–	may be	+	no
IC area overhead	+	no	–	yes
validity risks		IC test coverage		IC test coverage
			–	number of faulty ICs
			–	distribution of faulty ICs
			–	invalidation rule

Table 1.2: Advantages (+) and drawbacks (–) of WST methods

In my view the problems related with validity issues prevent diagnosis-based testing from becoming popular. The number and distribution of faults is not a crucial problem, because there exist diagnosis algorithms that give realistic bounds under any fault distribution for the correctness, provided that the number of faults does not exceed a certain limit. This limit is usually high enough and the possibly low yield of the IC technology does not endanger the validity.

However, all diagnosis algorithm require that the applied invalidation rule not be violated because of "malicious" faults in the dedicated circuitry. A valid diagnostic rule is even more important than good IC test coverage because of the following:

- A chip containing a fault that is not covered by the IC test will pass the test, but the diagnostic incorrectness for this chip does not influence the correct or incorrect classification of other chips.
- Since the diagnostic algorithm uses the adjacent chips as references, incorrectly diagnosed chips may involve that adjacent chips are also wrongly diagnosed. A comparator fault frequently implies the wrong diagnosis of some local chip even if the IC test is very good, and this local incorrectness can be propagated wafer-wide without limits.

In spite of this serious weakness, there are no diagosis-based WST approaches that are applicable for general ICs and take into account this validity risk.

1.2.4 Contribution in WST

This dissertation concentrates on fault-tolerant issues of the WST technique. A test session is proposed that is dedicated to the comparators and the comparison compression circuitry. It is formally proven that the test session detects any combination of multiple stuck-at faults in this circuitry. The diagnostic model is refined to include reliable and

unreliable comparisons. A wafer-scale scheme is proposed to meet the requirements of the dedicated circuitry test.

1.3 Environment and terminology

This section gives a short introduction to the terminology and some testing aspects of electronic circuits. A second goal of this classification is to set the scope of this work in the various test-related areas.

- *digital vs. analogue.* In digital testing, the values in the used circuit mode are have all discrete ranges. The analogue and continuous output measured during testing is quantised into discrete time and discrete values which must match exactly the expected values to pass the test. Digital testing has the property that the number of possible circuit states is finite. In *analogue testing* analogue-type outputs must remain within a specified range.
  - We deal with digital testing in this dissertation. We note that the data-dominated nature of BudaTest allows for a moderate extent of quantisation of analogue signals, although the size of the resulting problem space inhibits its use for larger circuits.
  - We propose the fault-tolerant techniques for digital WST. Diagnosis-based WST is applicable to analogue testing as well, but analogue comparators should be designed with care, and the presented fault-tolerant features apply for the digital part of the diagnostic circuitry.
- *physical vs. design faults.* Testing against design errors has many common features with testing against physical failures, but the set of design errors is not defined as exactly as in a physical fault model.
  - We always assume physical faults in this thesis.
- *permanent vs. transient or intermittent. Permanent* faults entail *deterministic* circuit behaviour in digital circuits while the circuit behaviour depends on whether or not *transient* or *intermittent* faults are present. Permanent faults are usually detected by dedicated post-manufacturing or maintenance off-line test sessions. Transient or intermittent faults are not detected by precalculated test patterns. Instead, fault-tolerant systems are protected against such faults by various on-line error detection mechanisms (watchdogs, error detection/correction codes, master/checker configurations etc.), which are methods based on information redundancy.
  - In accordance with these principles, the goal of the testing in this work is the detection of *permanent* faults.

- *deterministic vs. random TPG.* Practical ATPG usually starts with random TPG and the random phase is terminated when the generated random vectors do not detect enough untested faults. At this point, the only way to increase fault coverage is launching a deterministic ATPG algorithm which searches for test patterns using the circuit model and assuming specific *target faults.*
  - The BudaTest approach addresses efficiency issues of deterministic TPG.

- *detection vs. diagnosis.* There exist testing approaches aiming at *system diagnosis.* The goal of diagnostic tests is not merely the detection of the presence of faults, but the identification of the faulty part as well. They are typically performed in multi-component systems to find faulty components to replace, but they can be useful in the IC technology as well, inasmuch as they help identify critical design or manufacturing problems.
  - We do not address diagnostic issues in this work.
  - Note that the diagnosis-based WST approach is an interesting mixture of diagnosis and detection where the detection test of the chips is performed as the diagnostic test of the wafer. The proposed fault tolerant technique can be used by any wafer diagnosis algorithm.

- *functional vs. in-circuit.* According to the tester's access to internal points of the device, we can distinguish between *in-circuit* and *functional* testing (Figure 1.1). In in-circuit testing the tester can drive and observe the internal points of the tested circuit. Since components can be isolated and tested independently, the ATPG and testing tasks can be decomposed into several small and combinational tasks. In functional testing, the circuits are driven at their input pins and observed at their output pins, exactly like during normal operation. Since internal points are only indirectly controlled and observed, this latter type of testing requires longer and more complex test sequences.

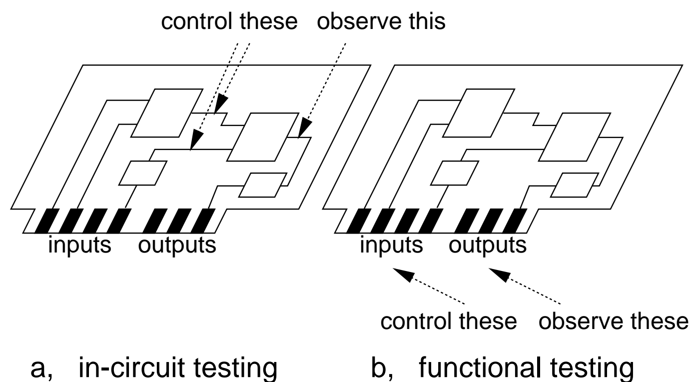

Figure 1.1: In-circuit and functional testing of a board

aspect	options
fault source	***physical***, manual design, transformation
fault life	***permanent***, transient, intermittent
circuit type	***digital***, analogue, mixed
ATPG kind	**deterministic**, random, pseudo-random
test purpose	***fault detection***, diagnosis
access type	***functional***, in-circuit
testing speed	at-speed (parametric), **static (functional)**
circuit age	*manufacturing*, maintenance

Table 1.3: Various testing aspects

A current trend in electronics is the increasingly frequent use of functional testing. In-circuit testing requires a costly ATE, and the rapidly growing speed and integration of devices raises other problems with the accessibility of internal points, especially in *surface-mounted* technologies.

There exist a few solutions by which a logically in-circuit test can be implemented as a functional test. *Design for testability* (DFT) is the comprehensive name of design solutions that provide a logical interface to physical access points. DFT techniques include test point additions and scan-based designs (including LSSD [3] and boundary-scan [4]). However, DFT is usually very expensive since it requires extra surface and additional circuit pins. Although for today's really large circuits the introduction of some DFT is inevitable, ATPG algorithms should be nevertheless improved and made capable of handling larger subcircuits, thus reducing the cost of DFT.

- Since test computation problems do not characterise the simple in-circuit tests, the BudaTest method should be used in an functional test enviropnment.
- The discussed WST technique is also a functional test, because it compares IC outputs.

Table 1.3 gives an overview about the listed and some other ATPG-related aspects. We use bold typeface when BudaTest is characterised by the given feature and italics when the feature is related to diagnosis-based WST.

1.4 Outline of the thesis

This work is structured as follows:

Chapter 1 presents the motivation of this work and lists what contributions it provides. It gives an introduction to ATPG-related terminology and sets the scope of the proposed approaches.

Chapter 2 describes the current trends and status of digital design. It describes the design process and explains its stages from the point of view of test generation.

Chapter 3 gives an overview on existing ATPG approaches. First, gate-level algorithms and their enhancement schemes are discussed. Then existing architectural and behavioural TPG approaches are presented together with their advantages, drawbacks and main application fields.

Chapter 4 describes what constraints are and how constraint satisfaction problems can be solved. It draws attention to the efficiency-related issues of the solving algorithms and gives an overview what main enhancement ideas the constraint literature provides. The chapter also presents an existing constraint-based gate-level ATPG approach.

Chapter 5 identifies modelling problems that must be solved if we want to raise the abstraction level of TPG. It shows how BudaTest handles these problems.

Chapter 6 presents the constraint solving engine incorporated in the BudaTest tool. It describes how high-level circuit information is exploited by the ATPG-specific solver, and presents performance data measured on high-level and low-level benchmark sets.

Chapter 7 deals with the fault-tolerant issues of the diagnosis-based WST approach. Since this chapter is not directly related to the previous chapters, a short introduction is given about the diagnosis literature. Then a specific circuitry test is proposed which is gradually extended until it covers all important parts of the diagnosis-related circuitry. A wafer template that allows for the execution of the fault tolerance test is also shown.

Appendix A describes the benchmark circuit set used for performance measurement.

Appendix B briefly introduces the structure and interface of the BudaTest program.

The dissertation is usually written in passive or plural first person. I use singular first person whenever I want to emphasise my personal contribution.

Chapter 2

Current trends in digital design

Since testing in inseparable from design issues, we summarise what major trends apply in digital design.

It is probably needless to emphasise in what extent the integration of circuit components improves, and, consequently, the number of transistors in a single unit grows. In the sixties and seventies when the ATPG methodology was developed, the terms *small, medium, large* and *very large scale integration* (SSI, MSI, LSI, VLSI), ranging from dozens to thousands of transistors on a surface unit, was a meaningful distinction between technologies. Now we speak of millions of transistors in a single chip, and this quantity doubles about every three years [5].

2.1 CAD tool features

With the increase in size and integration, design methodologies have undergone a fast development in the past ten years. It was initially the engineer's job to carry out the entire design procedure from the specification until the circuit lay-out. As the result of CAD methodology improvement, ad hoc designs of any complexity have been replaced by structured design techniques where the majority of tasks are automated. The methodology, implemented by state-of-the-art CAD tools (e.g. Cadence, Synopsys, Mentor [6, 7, 8]), are characterised by the following main features:

- *Hardware synthesis* is based on design libraries, which contain existing components provided by the CAD tool vendor or generated by the user. The libraries promote *intellectual property* (IP) reuse. Their use decreases the development time significantly, because the library elements do not need to be redesigned.

- The design direction is *top-down*. The goal of the digital design procedure is to gradually refine the initial specification into a structural description of utilised library elements. Accordingly, the design process consists of several stages represented by increasingly detailed and decreasingly abstract descriptions of the same circuit.

- The tasks that can be automated are passed from the engineer to the tool. As a result of a continuous improvement, the automatic entry point moves toward more and more abstract description levels. *Silicon compilers* that appeared about fifteen years ago transform a satisfactorily detailed structural description into lay-out plans. A recent improvement is the appearance of *high-level synthesis* (HLS) tools (e.g. AMICAL [9]) which push the automatic design entry point even higher in abstraction. They are capable of processing *behavioural* register transfer level (RTL) descriptions and generating *architectural* descriptions. The features of these levels will be discussed later in this chapter.

- A common hardware description language (HDL) is used at every stage of the design of a circuit. The most popular languages are the IEEE standard VHDL [10] and Verilog. VHDL has a high modelling power and allows for various descriptive styles: it permits the use of procedures and even of dynamically allocated memory where only behaviour is important, but it can be also used for the mass simulation of gate-level netlists. The advantages of VHDL include the portability between CAD tools and the possibility of mixed-level simulation.

- Tasks other than logic design are automated and incorporated in the CAD tool. Such tasks include automatic or computer-assisted verification (e.g. microprocessor verification in [12]), quality evaluation, or, as in the case of this thesis, ATPG.

2.2 Digital design flow

This section describes the stages of the design process which are important from the point of view of ATPG. The input of this process is the informal (and therefore ambiguous) specification of what the circuit should do, while the output is the lay-out masks directly used by the manufacturer. The intermediate stages are illustrated in Figure 2.1.

The first part of the process, denoted as *behavioural capture* in Figure 2.1 is actually a very complex manual task and usually consists of several substages and iterations. Large systems are first modelled at the system level by means of procedural and abstract constructs. This description can already be simulated but the used language constructs are so abstract that they cannot be automatically processed. Thus the description is manually refined and separated into smaller modules. During this transformation, many rules of thumb must be respected and the most abstract language constructs must be implemented in less abstract ways.

Since ATPG is related to the automatic part of the logic design, now we are not interested in this part. After the manual design, we reach the synthesizable behavioural level where automatic synthesis begins.

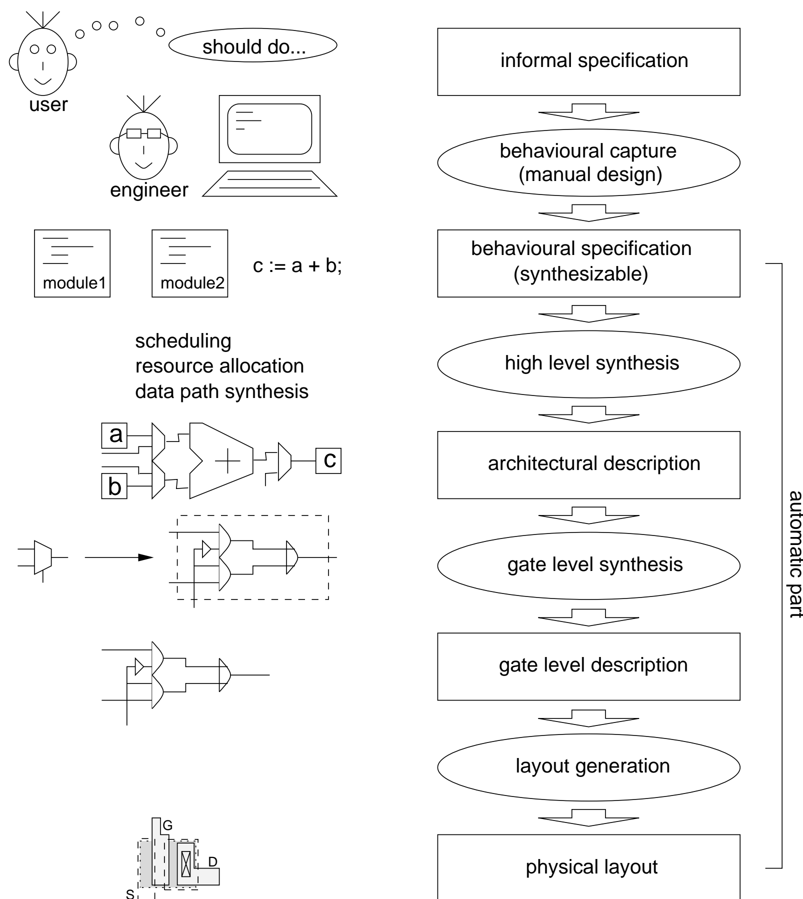

Figure 2.1: Digital design flow

2.2.1 Synthesizable behavioural VHDL

This level is characterised by the following features:

- The behaviour of a component is specified as a set of *VHDL processes*, which can be complex procedures as well. Certain restrictions apply to resetting and clock usage styles.
- The use of abstract data types and complex expressions is allowed.
- Conditional statements, loops etc. are permitted.

2.2.2 High-level synthesis

High-level synthesis tools read synthesizable descriptions and generate an *architecture*, i.e. a structure of high-level components. The main goal of HLS is to generate a structure *functionally equivalent* to the processes.

They go through the following basic steps:

- *scheduling.* The procedures in the behavioural descriptions are divided into time frames in this step. Instructions in a time frame will be executed simultaneously in one clock cycle.

  The HLS output is functionally equivalent to the synthesized behvaioural description, but its timing may be different. The code can explicitly define time frame boundaries with VHDL *wait statements*, but the HLS tool can also introduce new frames by revealing data dependencies or by other means.

  Assignment delays and other kinds of explicit time values of the behavioural specification are ignored. The exact parameters are determined by the characteristics of the library elements.

- *resource allocation.* VHDL operators and function calls are mapped into *functional units* (FUs), those components that implement the operations. There is a FU library whose elements are capable of executing one or more VHDL operations. An *adder* unit, for example, implements the "+" operation while a suitable ALU can be utilised wherever the VHDL parser encounters the "+", "-", and the word-wise logical operators.

  Scheduling and resource allocation are complementary tasks. There are usually design options to prefer either fast or cost-effective (in terms of surface usage) design styles. A cost-driven design, for example, may define two time frames for instructions which use the same FU resource and would be executed parallelly in a speed-driven design.

- *data path synthesis.* This step generates the wires and defines the routing between data containers and FUs. In a *multiplexer-based* design, multiplexers are placed in front of multiply used FUs and before data containers that are driven by several assignments in a procedure. In a *bus-based* design the multiplexer outputs are replaced by buses and multiplexers are substituted with bus driver auxiliary elements.

- *control flow extraction.* The HLS tool creates a so-called *control part* in the form of a behaviourally described finite state machine (FSM). This component collects the Boolean signals coming from the data part and generates control signals such as the selector inputs of multiplexers, register write enable signals and operation codes of multi-operation FUs.

It is usually possible to define design constraints that limit the number of time frames, the circuit area, the power consumption and other various design parameters. If such a constraint is violated, a subsequent iteration may produce a different architecture.

2.2.3 Architectural VHDL

We call the HLS output description style *architectural*. Its main features are the following:

- It is a *structural* description, i.e. a list of components interconnected by signals.
- The signals have abstract data types (integer, bit vector, or even records) and therefore large word widths.
- The components correspond to the elements of the FU library, can be high-level and sometimes sequential. In addition, there are registers needed for data storage, and auxiliary components required for the establishment of data paths.
- The components can be parametrised. The used instance depends on the actual data size.
- Control and data are clearly separated (Figure 2.2). The signals within the data part are data signals and may have wide abstract types. The signals interconnecting the two parts are considered as control signals. Thus control signals include expression result signals, FU operation codes, multiplexer selectors (bus driver controls), and register control signals. Control signals have a small range.

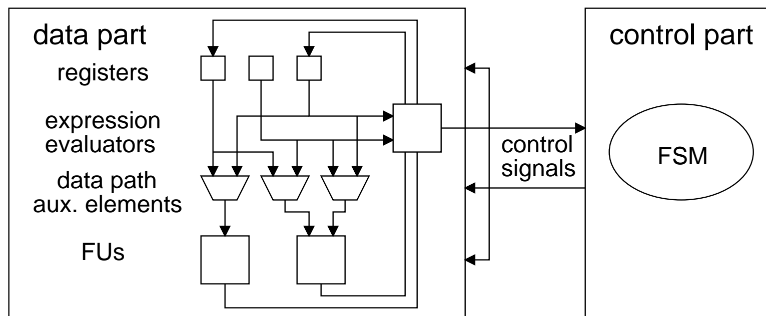

Figure 2.2: Architectural style

2.2.4 Low-level synthesis

Low-level synthesis (LLS) starts with an architectural description and generates the target of the design, usually the lay-out plans and the masks required in manufacturing. The most important tasks are the following:

- *component and wire placement*. This task determines the physical position of the pre-designed library elements and the wires interconnecting the components. In addition to the overall surface optimisation, a number of design rules must be respected to avoid technology-related problems.

- *control part synthesis.* The FSM generated by the HLS tool is implemented, e.g. with a PLD.
- *parametric design*, e.g. decisions on the resolution of imaging, design for power consumption etc.

From the point of view test generation, the most important feature of LLS is that decisions made during LLS do not influence the logical manifestation of the most widely assumed physical faults. A stuck-at fault (see Section 3.2.1), for example, is defined the same way for any line width and wire placement.

2.2.5 Gate-level descriptions

LLS is not necessarily divided into *gate-level synthesis* and *lay-out generation* as shown in Figure 2.1, because the vendor-supplied library elements can directly contain lay-out information. We still discuss this intermediate stage, because it is important from the point of view of traditional ATPG methods.

Gate-level descriptions are the last logical and technology-independent representations of a digital circuit. The characteristics of this level are as follows:

- It is a structural description.
- Only the *bit* data type exists. Originally complex signals are encoded into a set of bit-type signals.
- The component "library" is fix, containing the basic gate types *(and, or, xor, nand, nor)* and basic flip-flop types (*D* primarily).
- The number of signals and components is enormous. It is usually hopeless to understand the function by human reading.

Table 2.1 gives a summary on some important features of the different representation levels and description styles.

	system-level	behavioural	architectural	gate-level
simulatable	yes	yes	yes	yes
synthesizable	no	yes	yes	yes
procedures allowed	yes	yes	no	no
components allowed	yes	yes	yes	yes
component type	any	any	library element	gate
data type	abstract	abstract	abstract	bit
compact	yes	yes	yes	no
control-data separation	no	no	yes	no

Table 2.1: Features of representation levels

Chapter 3

Previous ATPG approaches

This section gives an overview on existing ATPG approaches for digital circuits described at various levels. We divide the discussion into presenting *gate-level* and *high-level* approaches.

3.1 Fault models

I devote special attention to the question of fault modelling because in my view this determines the abstraction level that can be used for ATPG.

Faults are physical failures which are caused by defective hardware material or production machinery, by incorrect design or by human error (Mourad [13]).

Logical faults change the logical function of the circuit while *parametric faults* modify non-digital circuit parameters such as delay, capacitance, temperature dependence etc. [1].

The *fault model* is the representation of the physical fault effect in a given application. Fault modelling must therefore conform to circuit modelling some way so that the fault effect can be interpreted at the level the application uses. Obviously, the relation between physical faults and the fault model must be thoroughly examined in every application that uses fault models. In particular, the following things must be proven:

- Physical faults indeed cause such a perturbation as assumed by the fault model.
- The fault model is able to represent all the expected kinds of physical faults. Certainly, we cannot take into account every imaginable fault, but the most likely ones should be listed.

3.2 Gate-level ATPG

Gate-level ATPG algorithms have existed since fairly complex digital circuits appeared in the sixties. We show what basic gate-level methods exist and what improvements

have been made to traditional algorithms. Besides this overview, a good comparison of gate level methods can be found in [27].

3.2.1 Gate-level fault model

The most popular logical fault model is the *stuck-at (s-a)* model. This fault model assumes that physical faults are manifested in the logical form that certain signals fail to hold a 0 or 1 logical value. A signal stuck at 0 (1) always carries 0 (1) even if driven with 1 (0) by a certain gate.

Though the stuck-at model is defined as signal problems, the analysis of different technologies shows that it represents fairly well component (gate) faults as well. Many internal transistor faults are equivalent to the logical s-a fault of some signal appearing in the gate-level description. We note that there are a few transistor faults in MOS technologies that cannot be represented as stuck-at faults. There exist other dedicated fault models (*stuck-open, stuck-on*) to cover these problems [14, 15], although they are rarely used in practice.

The *single stuck-at* fault model allows at most one stuck-at signal. The *multiple stuck-at* model allows several s-a faults to occur at the same time but the fault list is very long in this case.

In the *short* fault model (also known as *bridging* or *coupling* faults) two signals fail to hold different values. If they would take different values in the fault-free case, the common value is determined by the used technology. Certain technologies cause *wired-or*, others cause *wired-and* relation between the shorted signals.

Figure 3.1 shows an activated s-a-1 and a wired-or short fault. Both faults cause an erroneous 1 value instead of the correct 0 at the fault location. Note that the input patterns are test patterns as well, since the circuit without the fault would produce a different response.

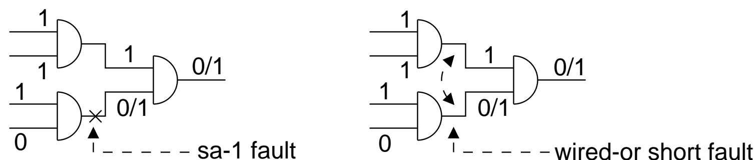

Figure 3.1: Active s-a-1 and wired-or short fault

Extensive studies show that test pattern sets generated for single s-a faults cover fairly well other fault types [16]. Applications that use the single s-a fault model are usually considered by the testing community as accurate in fault modelling.

3.2.2 Gate-level ATPG algorithms

In the following, *functional* TPG algorithms will be presented. Such an algorithm finds test patterns for which the logical function modification effect of the fault becomes

composite value	fault-free value	faulty value
0	0	0
1	1	1
D (discrepancy)	1	0
$\bar{D}$ (not D)	0	1

Table 3.1: Meaning of composite values in the D algorithm

input 1	input 2	output
0	0	0
0	1	1
0	D	D
0	$\bar{D}$	$\bar{D}$
1	0	1
1	1	1
1	D	1
1	$\bar{D}$	1
D	0	D
D	1	1
D	D	D
D	$\bar{D}$	1
$\bar{D}$	0	$\bar{D}$
$\bar{D}$	1	1
$\bar{D}$	D	1
$\bar{D}$	$\bar{D}$	$\bar{D}$

Table 3.2: D propagation table of an OR gate

apparent.[1]

The D algorithm

The first systematic ATPG approach was the D algorithm presented by Roth in 1966 [17]. As every systematic ATPG method, it is a composite simulation of the fault-free and faulty circuit. He used the 4-valued logic of $\{0,1,D,\bar{D}\}$ to model value pairs (Table 3.1).

The gate pairs of the two simulations are handled together in a similar way. The truth table of and OR gate is given as an example in Table 3.2.

The D-algorithm consists of three basic phases:

[1] There exist other kinds of testing which exploit parametric effects of faults. I_{DDQ} *testing*, for example, is based on the fact that an activated fault is likely to involve abnormal power consumption, and does not require that the fault effect be functionally visible on any output.

1. *fault sensitisation.* An activated fault is a prerequisite for successful test generation, thus this phase assigns D or $\bar{D}$ to the faulty signal pair. The input signals of the gate driving the fault location are set to produce the negated s-a-value.

2. *D propagation.* A path is selected from the fault location to one of the outputs. The other input of the gates comprising the path are set in a way that the D or $\bar{D}$ value can be propagated.

3. *justification.* The remaining signals are set backwards by using the gate truth tables (without the tuples containing Ds or $\bar{D}$s).

Since contradictions may occur, backtracking is allowed in all phases. The D algorithm does not define priorities between truth table rows when more options are available, so decisions can be regarded as random. This can lead to an unnecessarily high number of backtracks. Another problem is backtracking in the D propagation phase. If the path selection is contradictory, another D-path must be chosen. When the fault location is far from the output, the number of paths can be very large.

PODEM

Goel's PODEM [18] is the most frequently used gate-level ATPG algorithm. It is a rearrangement of D steps where decisions consist in assigning values to the input signals. Once a new input is set, the circuit is partially simulated as deep as possible.

The goals are again fault sensitisation and D propagation. The ordering of variables to assign reflects these goals. The so-called *backtrace* technique is used for selecting the variables that have the largest impact on the *target signals*. The first target signal is the fault location, then the signals constituting a selected D-path to the output are targeted. A very important feature of the algorithm is the use of *controllability measures* during backtracing and of *observability measures* during D-path selection. These measures were proposed by Goldstein [19].

The input-oriented nature of PODEM means that it is efficient for circuits with relatively few inputs, but its performance drops with the increase in the number of inputs.

Composite justification

Proposed by Sziray [20] in 1979, *composite justification* consists of output selection and only *justification* steps. Once an output is selected for fault effect observation, the algorithm assigns D or $\bar{D}$ to the output and the fault location. The rest is simply backward justification using the full tuple set of the gate truth tables, during which the signal values become determined from the output to the inputs. Since the full tuple set contains Ds and $\bar{D}$s, the circuit is first preprocessed in order to explore where discrepancies are admitted and what signals matter at all. This filtering technique is called *node classification.*

From the point of view of this work, composite justification has similar features with the approach proposed here. Although Sziray does not use constraint terminology, the lack of the path selection phase makes the entire representation uniform. Furthermore, the *node classification* technique is the first explicit technique that reveals which signals may take different values in the two simulation. The *colouring* technique presented in Section 6.4 has the same goals, although the implementation technique is rather different.

3.2.3 Implication

The FAN algorithm (Fujiwara, [1]) is an improvement to D. It performs an *implication* step to explore and effectuate the value assignments that are equivalent with the most recent decision. The result of implication is a smaller subspace of unassigned wires after every decision, which necessitates fewer backtracks.

The idea of assigning implied values to signals other than the decision subject appears most explicitly in the 16-valued approach of Hegedüs [21]. Each value used by his algorithm represent a superset of the basic set $\{0,1,D,\bar{D}\}$. This greatly improves the implication capabilities of the ATPG tool because the exclusion of every basic value combination can be expressed. The drawback of the method is the large size of gate descriptors. Similar 9 or 10-valued logic is used in [22] and [23].

Another very important result of Hegedüs' work is the decentralised implementation of the ATPG decision engine. The mechanism of *weights* that can be associated to decision choices provides good support for topological measures an heuristics. He has shown that the generic algorithm may become PODEM or other algorithms by means of extreme weighting, so the algorithm can match the characteristics of the circuit it is applied to.

3.2.4 Sequential extension

The basic gate-level algorithms generate single-vector tests for the faults of combinational circuits. Sequential circuits may have (and usually do have) faults which require not test vectors but sequences to detect.

There is a simple ATPG extension technique that enables combinational algorithms to cope with sequential circuits. The *iterative array* model [43] cuts the feedback loops of the sequential circuit and unrolls the consecutive time frames into a large combinational circuit (Figure 3.2 and Figure 3.3). Finding a test sequence for the sequential circuit is equivalent to finding a test vector for the unrolled circuit.

The iterative array technique is applicable when the following circuit requirements are met:

- The concept of "consecutive time frames" requires synchronous single-clock sensitivity.

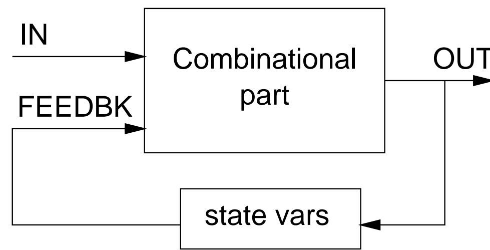

Figure 3.2: General sequential circuit

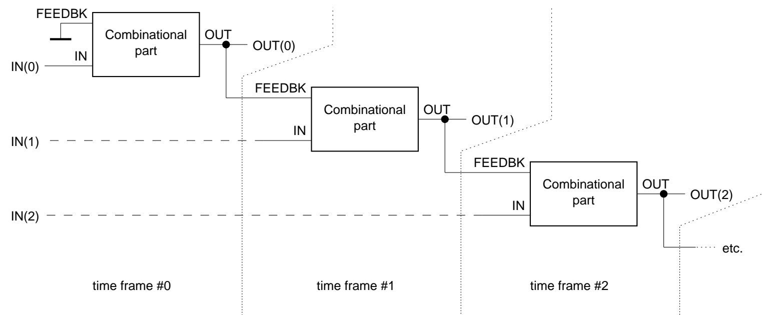

Figure 3.3: Iterative combinational model of a sequential circuit

- Registers must have *reset* so that their initial value be known to the ATPG algorithm in time frame 0.

Although there are proposals in the literature aiming at the testing of asynchronous circuits without known homing sequences, this problem is rather avoided than solved.

The price of the extension is the soaring size of the ATPG problem space. The problem space of an n-frame long expected test sequence is n times larger than that in the combinational case, which is crucial in an NP-complete problem.

3.2.5 CONTEST

CONTEST (Tilly, [25, 26]) is a gate-level tool that solves ATPG as a *constraint satisfaction problem* (CSP). We give special attention to CONTEST because the BudaTest approach presented in this work is based on the same constraint-based principle as CONTEST. CONTEST's main features are the following:

- The signal pairs of the fault-free and faulty simulation are regarded as *constraint variables*. (Section 4.1 or [27, 25] contains constraint-related definitions.)

- The function of the gate pairs is described as *constraints* over the variables.

- The variables have an initial domain of $\{0,\ 1,\ \mathrm{D},\ \bar{\mathrm{D}}\}$. The actual domain of a variable describes what values can be part of a solution, which is as powerful in implication as the 16-valued logic of Hegedüs [21]. Variables are assigned locally consistent values by decisions.

- The requirements against a test pattern are expressed by the initial domains of the variables. The initialisation of CONTEST is practically the same as that of *composite justification* [20], i.e. the fault location and a selected output is initialised with $\{D, \bar{D}\}$, and a domain preprocessing similar to *node classification* is performed.
- Since the field of CSP solving has an already wide literature, general CSP solving ideas are employed to speed up the ATPG process. In particular, a variation of *dependency-directed backtracking (backjumping)* [59] is used.
- Since all ATPG requirements are represented as constraint network data, the CSP solver has a high extent of freedom in decision control. In CONTEST, a *hesitation queue* is established to store decision candidates. This helps keeping decisions concentrated in one growing part of the circuit, but also leaves space for measure-based heuristics.

3.2.6 Algorithm evaluation

Since there are no analytical features to which the circuits designed in practice conform, the performance evaluation of ATPG algorithms is performed by *benchmarking.* Besides the benchmark circuits that individual approaches propose as measurement basis, there exists a commonly accepted gate-level benchmark circuit set. The ISCAS benchmark [11], specially designed for testing gate-level related tools, contains circuits different in various aspects: different input/internal/output signal numbers and ratios, different gate numbers and ratios, tree-structured vs. highly reconvergent etc.

3.3 High-level ATPG approaches

High-level approaches are characterised by an abstract model which describes the propagation and dependencies of usually abstract data. We will call *architectural* those ATPG approaches where the abstract model is in direct correspondence with the actual circuit hardware, and *behavioural* the other high-level approaches.

3.3.1 Architectural approaches

Extensions of gate-level approaches

Many gate-level algorithms use some combination of three basic steps (fault sensitisation, D propagation, justification). There is no theoretical obstacle in defining high-level equivalents of these steps for architectural descriptions. The new methods must revise the following aspects:

- In contrast to gate-level approaches which handle the two simulations as one by defining composite values (see Table 3.1), abstract value pairs cannot be efficiently

treated as one value.

- Value tables either become very large or must be replaced with the execution of functional component procedures. Backward justification, i.e. the inversion of component procedures becomes rather difficult. For this reason, high-level PODEM which uses only forward data propagation is easier to implement [42].

- Fault modelling becomes nontrivial. Single-bit stuck-at faults, for example, affect single bits of abstract values, the effect of which must be separately defined. Internal faults of components must also be taken into account.

Many existing approaches belong to this category. The D algorithm was defined for hardware description languages in [24]. An extension for the 9-valued logic gate-level approach [23] was done by Steingart et al. in [28]. The DIAS tool [29] performs high-level composite justification (Section 3.2.2, [20]) for a dedicated hardware description language OPART [30].

In fact, the BudaTest method described in this work can be also regarded as the extension of the CONTEST constraint-based gate-level ATPG approach. Chapter 5 deals with modelling problems that are implied by the use of abstract values and high-level components. However, the extension consists not merely in the transformation of steps into abstract equivalents, but in the use of abstract algorithms which have no gate-level equivalents.

S-graph

S-graph based circuit representation was used by Thatte and Abraham [31], and was refined by Brahme and Abraham in [32]. The method is specific to microprocessors whose register-level model is the *system graph* (S-graph). The nodes of the S-graph represent the registers of the processor, while edges between nodes stand for instructions that move data (manipulated or not) between the registers. Two distinguished nodes, IN and OUT, model the pins controllable and observable by the external word.

The processor is regarded as a set of the following functions: *register decoding, instruction decoding and control, data storage, data transfer*, and *data manipulation.* A functional fault model is defined for each of the specified functions except data manipulation.

The faults of the register decoding, data storage and data transfer functions are tested by relatively simple algorithms which are mainly based on the topological distance of the registers from the INPUT and OUTPUT nodes. The instruction decoding and control function model was later refined in [32] and its functional faults were tested by assigning codewords to registers.

The S-graph approach is an efficient architectural method for testing general parts of microprocessors. The test generation time and the test length is polynomial with the number of registers. However, data manipulation faults are not considered and

microprocessor features are heavily exploited, thus the method is hardly extendable to general digital circuits.

The S-algorithm proposed by Su et. al [33, 34] uses a similar but somewhat more general model for register transfer level descriptions. The method still assumes some explicit instruction sequence.

Alternative Graph

Ubar's method creates *generalised decision diagrams* based on the circuit and fault description [36].[2] Unlike in traditional algorithms dealing with *binary decision diagrams* [35], high and low abstraction levels as well as control and data faults are efficiently and uniformly handled by generalised DDs.

A novel concept of mixed level combining of deterministic and random techniques in test generation is introduced in AG. On the RT-level, deterministic path activating is combined with constraint techniques by means of random techniques. The gate-level local test patterns for components are randomly generated driven by high-level constraints and partial path activation solutions.

The AG approach permits test generation for finite state machines (control parts) as well [37]. For the description of functions, structure and faults in a FSM, three levels are used: functional level (state transition diagrams), logical or signal-path level, and gate level. For all these levels a uniform description language, a uniform fault model and uniform procedures for ATPG and test analysis were developed. This uniformity allows easily to move and carry partial results from level to level when solving the mentioned tasks.

The path activation and the FSM testing features of the AG approach make it an excellent method for control-dominated circuits where long and non-trivial paths and state transitions. High data dependencies are, however, not necessarily handled effeiciently due to the randomness of the solving of such subtasks.

3.3.2 Hierarchical testing

Hierarchical testing is not an alternative to architectural approaches but an additional technique which allows the accurate modelling of gate-level faults.

In architectural approaches *components* and *interconnections* are meaningful terms because they reflect the circuit architecture. The different high-level ATPG algorithms generate tests for interconnection faults that appear in some way in the utilised circuit model. Thus high-level algorithms do not directly cover component faults.

High-level testing is introduced on account of the expected performance advantage due to the circuit model compactness. However, the effect of component faults cannot be generally described at the high level, because modelling physical fault effects by altered functionality is unrealistic. Thus we face two contradicting requirements:

[2] Generalised decision diagrams were first poroposed under the name of *Alternative Graphs* (AG).

1. High-level representation is required for efficiency.

2. Accurate models of component fault effects can be obtained only by low-level component representation.

Hierarchical testing offers a reasonable solution for this problem by means of *mixed-level modelling*. In a high-level tool supporting hierarchical testing the components have two representations:

- a high-level representation *(functionally described)* for efficiently representing the fault-free operation. For example, an adder component can be represented with the high-level addition operator.

- a low-level representation *(structurally described)*, consisting of several smaller components for exact fault modelling. For instance, the same adder component can be represented with a chain of full adders. The low-level representation must conform to the design generated by the CAD tool to obtain exact fault effects.[3]

In the mixed-level technique, only the component affected by the assumed fault is represented with the low-level model, while others are modelled with the high-level equivalent (Figure 3.4). This way the low-level representation of certain components does not entail a dramatic increase in the component number, and the average abstraction level drops only in a minimal extent. The hierarchical principle can be applied in a recursive manner (Figure 3.5).

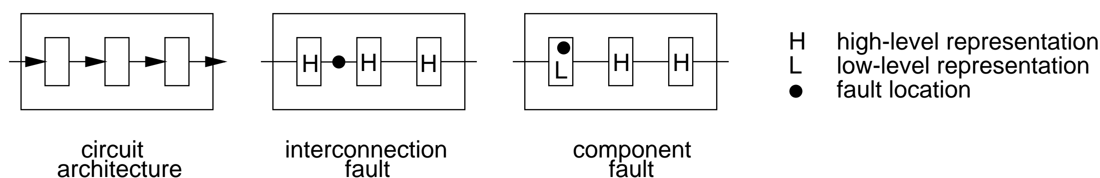

Figure 3.4: Use of high and low-level models

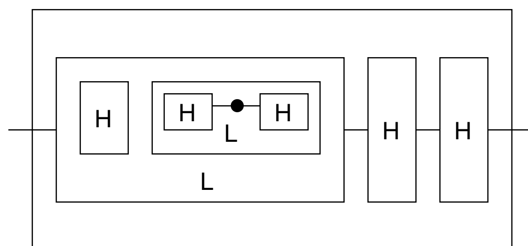

Figure 3.5: Recursive application of the hierarchical principle

[3]In theory, *functional replacement* would be an alternative way to model faulty behaviour with preserving efficiency. However, as it is argued in Section 3.3.3, only a small fragment of realistic physical fault effects can be described at the high level, therefore the applicability of this concept is limited.

3.3.3 Behavioural approaches

We call *behavioural* those approaches where the circuit graph (or whatever other model is used) is extracted from the procedural HDL description of the circuit. Such ATPG methods are the Behavioural Test Generator (BTG) of O'Neill et al. [38] defined for general HDLs, and other algorithms developed from it. Cho and Armstrong suited the algorithm to the semantics of VHDL [39]. They presented later the B-algorithm [40], which can generate tests for behavioural faults, and follows the basic steps of the D algorithm with good/bad value pairs and high-level VHDL assignment statements.

The fault model of these approaches is defined as the perturbation of the HDL source code of the circuit. The behavioural modification includes the following:

- *signal assignment faults.* A VHDL signal assignment statement assigns the value of an expression to a signal. A behavioural stuck-at fault is a stuck-at fault of a bit of a signal or a *virtual signal.* Virtual signals are intermediate signals used for expression construction, or fan-out stems and branches when a signal is used in several expressions. The fault model allows so-called *behavioural stuck-open faults* as well for incorrectly performed assignments.

- *control faults.* A control fault is the incorrect execution of a VHDL conditional statement. Control faults include stuck-then, stuck-else branches, dead clauses, dead processes etc.

- *microoperation faults.* An arithmetic or relational VHDL operator is faulted to another operator. For example, the faulty model may execute addition instead of subtraction.

In the view of the author, the price of compact circuit modelling is improper fault modelling in these approaches. Signal assignment and control fault modelling suffers from ambiguity.

Section 2.2.2 describes how high-level synthesis works and what decisions it makes. We show how different structures HLS may produce from the same behavioural description. Even if bus-based and multiplexer-based architectures are rather similar, the ambiguity originates in different *scheduling* and *functional unit allocation* strategies. If the VHDL code contains n "+" operations, specifying strict area constraints for the HLS tool may result in allocating only one adder component and introducing a separate time frame for each addition *(FU reuse).* On the other hand, preferring a quick architecture could allocate n adders, reduced by the number of concurrently performable operations, which may depend again on whether the HLS tool is sophisticated enough to recognise parallellism.

Example: Consider the following VHDL code:

```
if x > y then
```

```
    x := x - y;
  else
    y := y - x;
  end if;
```

A sophisticated tool notices that one subtractor FU is sufficient for this fragment, because they stand in mutually exclusive branches of an *if statement*. It may generate therefore a data part with a single subtractor FU and a condition evaluator FU. A less intelligent tool could decide that two FUs implement the two subtractions. (In fact, the same tool can decide so if there are other subtractors used in other time frames.) Combined with bus-based data path synthesis, the same VHDL code could result in an entirely different the data part implementation (Figure 3.6). As a consequence, exact component fault effects are unknown in the behavioural stage.

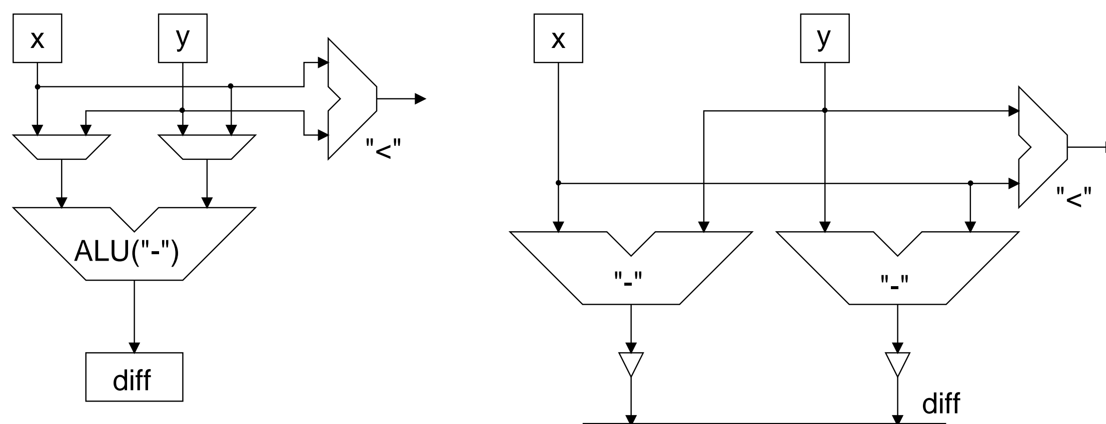

Figure 3.6: Different architectures of the same behavioural source

Of course, where the behavioural test generator is part of a CAD tool whose high-level synthesis algorithm is known to the ATPG tool, this fault modelling technique (except microoperation faults) is admissible. In this case, however, the *behavioural* fault model is nothing else than a reverse-engineered *architectural* fault model. It is easy to show the equivalence between behavioural signal assignment faults and architectural component interconnection faults, as well as that between behavioural control faults and architectural control signal faults. Some behavioural microoperation faults may have an architectural *opcode s-a fault* equivalent, but it is quite speculative to claim that general physical faults may cause arithmetic units to perform other operations. Instead, the space for component fault modelling should be left open, as it is done in architectural approaches.

Although the fault model employed by the behavioural approaches covers physical faults of some signals that would appear in the top-level hierarchy of the subsequently generated structure, some features, e.g. microoperation faults, are rather applicable to *design faults*.

3.4 BudaTest objectives

The summary and the consequences of the previous discussion of ATPG methods are shown in Figure 3.7. We regard the architectural level as the highest possible ATPG entry point when physical faults are concerned. An earlier ATPG would entail unrealistic modelling of physical faults, while a later ATPG start would cost prohibitively much ATPG time. Similar conclusions are reached in Benyó's ATPG comparison [41].

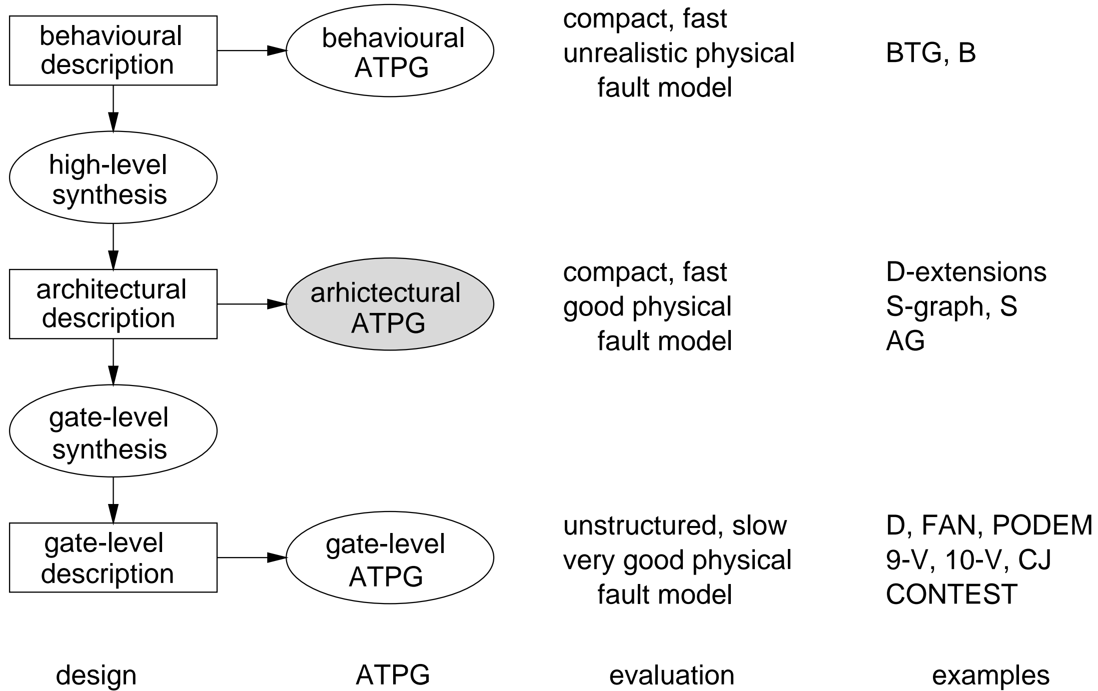

Figure 3.7: Comparison of ATPG entry points

In spite of the existence of architectural ATPG approaches, there is a lack of methods that could efficiently handle general data-dominated high-level circuits. Although D-extensions equipped with mixed-level testing are theoretically capable of handling such circuits, they contain no new invention that would exploit high-level features. The S-graph and S-algorithm approaches are fast functional tools where the correspondence between functional faults and physical faults is apparent, but their use is limited to microprocessors or microcontrollers, and they do not deal with data manipulation. Ubar's AG approach is promising for general circuits but especially for control-dominated ones.

The BudaTest approach, presented in this work, is intended to handle data-dominated circuits with intensive data manipulation and moderate control. *Moderate control* means that the number of control signals is unrestricted but the *control part FSM* (see Section 2.2.3) can be implemented with a few gates.

BudaTest is not fine-tuned yet as long as heuristic decision control is regarded. We will show, however, that the application of non-heuristic methods that extensively exploit the high-level characteristics of the architecturally described circuit brings such

improvement in terms of fault coverage and time demand that makes it a competent high-level tool. In addition, it solves such modelling problems as the handling of vector slicing and indexing and other practical issues.

Chapter 4

Constraint-based modelling

The concept of discrete constraints provides a convenient means to describe the various requirements for an input pattern to be a test pattern for a given fault and a given circuit. The operation of the digital circuit, the effect of the considered fault, and the fault effect observability condition can be all represented as constraints.

4.1 CSP definition

A *constraint network* is a set of *constraint variables* and *constraints* defined over the variables (Dechter, [52]).

The *constraint variables* (or simply variables) $X = \{X_1, \ldots, X_n\}$ are defined by their domains $D = \{D_1, \ldots, D_n\}$. The D_i domain of the X_i variable describes the set of the x_{ij} values X_i can take.

A *constraint* C_i is defined as a k-ary *relation* over k variables, which is expressed as a subset of a k-ary Cartesian product:

$$C_i \subseteq D_{i_1} \times D_{i_2} \times \ldots \times D_{i_k}$$

Less formally, a constraint permits some consistent value combinations of the involved variables and excludes the others. A constraint C_i is *satisfied* when the relation evaluates to *true* for a value assignment $\{x_{i_1 j_1}, x_{i_2 j_2}, \ldots, x_{i_k j_k}\}$, i.e. when the tuple is an element of the subset defined by C_i. We also say that the $\{x_{i_1 j_1}, x_{i_2 j_2}, \ldots, x_{i_k j_k}\}$ tuple is *locally consistent* with respect to C_i.

A value assignment $x_s = \{x_{1 j_1}, x_{2 j_2}, \ldots, x_{n j_n}\}$ ($x_{i j_i} \in D_i$) is a *solution* of the constraint network if all the constraints are satisfied with the corresponding subsets of x_s. x_s is also said a *globally consistent* vector.

The *constraint satisfaction problem* (CSP) is to find a solution (or all solutions) for a given constraint network. A CSP is in general NP-complete (Montanary, [49]).

A constraint network is called a *binary constraint network* if every constraint is defined over at most two variables. The largest part of the existing constraint literature has been elaborated for binary CSPs.

A *constraint graph* is the graphical representation of a constraint network. A binary

constraint network can be represented by an ordinary graph $G : \{X, C\}$ possibly containing loop edges. Each node of G represents a constraint variable X_i and there exists an edge between X_i and X_j if there is a binary constraint over X_i and X_j. There exists a loop edge around X_i of G if there is unary constraint defined over X_i. The constraint graph of a non-binary CSP is a so-called *hypergraph* where *hyperedges* connect more than two nodes.

We will call the Cartesian product $D_1 \times D_2 \times \ldots \times D_n$ the *state space* of the CSP. Thus, a solution is an element of the state space allowed by the constraints. Similarly, a *state subspace* is the Cartesian product of the ranges of some constraint variables.

Examples

The following is a CSP of three variables: X_1: (a, b); X_2: (c, d); X_3: (e, f). Let C consist of three constraints:

$$C_1(X_1, X_2) : (ac, ad, bc), C_2(X_1, X_3) : (af, be, bf), C_3(X_2, X_3) : (ce, de, df)$$

The CSP has two global solutions: *adf* and *bce*, out of the state space of eight tuples. In this example, the constraints were given by means of enumeration of the allowed pairs.

The next exemplary CSP has three variables, X_1, X_2, and X_3, each having a domain of integer numbers between 0 and 9. Thus the whole state space consists of 1000 tuples. The equality system

$$2 * X_1 = X_2 \qquad X_2 * X_2 = X_3$$

can be regarded as two implicitly given constraints over the variables, because the two equations define *relations* in $X_1 \times X_2$ and $X_2 \times X_3$, respectively. The CSP has two solutions:

$$X_1 = 0, X_2 = 0, X_3 = 0 \text{ and } X_1 = 1, X_2 = 2, X_3 = 4$$

A CSP does not necessarily have a solution. If we add a new constraint $X_1 + X_2 > X_3$ to the latter CSP, we obtain the CSP shown in Figure 4.1.[1] This CSP is unsolvable. Although every individual constraint can be satisfied with a properly chosen value pair or triple, all the constraints at the same time cannot be satisfied. This shows that *locally consistent* value sets do not necessarily lead to a *global solution* of the overall problem.

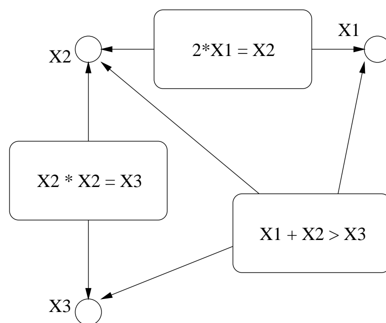

Figure 4.1: Constraint network hypergraph representation

[1] Rounded boxes denote hyperedges.

Digital gates, e.g. a NAND gate can be also modelled with constraints. A 2-input NAND gate defines a relation between three variables, two of them corresponding to gate inputs, one corresponding to the gate output. In fact, this is the modelling technique on which Tilly's CONTEST ATPG system (Section 3.2.5 [25]) is based, although it defines one constraint for the gates of the fault-free and faulty simulations.

Using $\{0, 1\}$ logic, the problem space consists of 8 tuples and contains all possible value combinations. Since the NAND gate output is a function of the input signals, only the following four them are consistent: 001, 011, 101, 110 (the third value in a tuple corresponds to the output).

4.2 CSP solution

Since discrete CSPs have finite state spaces, the solution can be found by exhaustive search. The question is therefore not whether the CSP can be solved but how this can be done efficiently. As the CSPs derived from practical problems (including ATPG) often have huge state spaces, it is essential that the solution (or one of the solutions) be found quickly.

4.2.1 Backtracking

In order to find an existing solution, a CSP solver algorithm must proceed in a *systematic* (often called *deterministic*) way through the state space. In the CSP literature the *backtracking* search technique is universally used to guarantee the systematic feature due to its limited space complexity. The approach presented in this work is also based on backtracking.

Backtracking is conditional searching. Its basic steps are assigning some values to some variables *(decision)*, then looking for a solution under the assumption the assigned values make part of the solution. If the resulting subproblem turns out to be unsolvable under this assumption, the assumption, disapproved by the subsequent search, is withdrawn, and other assignment is tried. This principle can be applied in a recursive way for the resulting subproblem. Decisions are also called *forward steps*, while decision withdrawals are called *backward steps* or *contradiction resolutions.*

In Dechter's definition [53] a decision is an assignment to a single variable. In this case the backtracking algorithm proceeds as follows:

Forward($x_1, \ldots, x_i$)
  if $i = n$ **then** exit with current assignment
  $C_{i+1} \leftarrow$ **ComputeCandidates**($x_1, \ldots, x_i, X_{i+1}$)
  if $C_{i+1} \neq \emptyset$ **then**
    $x_{i+1} \leftarrow$ first element in C_{i+1}
    remove x_{i+1} from C_{i+1}

```
        Forward( x_1, ..., x_i, x_{i+1} )
    else
        Backward( x_1, ..., x_i )

Backward( x_1, ..., x_i )
    if i = 0 then exit. No solution exists.
    if C_i ≠ ∅ then
        x_i ← first element in C_i
        remove x_i from C_i
        Forward( x_1, ..., x_i )
    else
        Backward( x_1, ..., x_{i-1} )
```

The **ComputeCandidates** function selects all values for an X_i variable that are consistent with previously assigned values. Backtracking is started with $i = 0$ and a **Forward**() call.

4.2.2 The decision tree

The operation of backtracking can be easily illustrated by the *decision tree* (DT) notation.

The full decision tree (FDT) is a tree graph which describes the order in which the state space is built up. The arcs of the tree represent decisions. When two nodes are related as parent and child, then the child represents the result of a decision made in the parent state which transfers the system in a more determined state. The root node is the initial, entirely undetermined state, while the leaves stand for final states where every variable is assigned. The leaves where the assigned values satisfy all constraints are the solutions of the CSP. Note that the structure of the FDT is highly algorithm-specific, depending on the nature and order of decisions. The set of the leaves of the FDT is always the problem space of the CSP while their number is always $\prod |D_i|$, the production of the domain cardinalities.

Since backtracking is *depth-first search*, the backtracking algorithm traverses the nodes in depth-first order. When it visits a node, it checks whether the currently determined variables are locally consistent or inconsistent. In the former case, it is either a FDT leaf (i.e. a CSP solution) or a new node creation (a forward step) follows. Otherwise, backtracking goes back to the parent node (a backward step) by means of retracting the decision the arc represents, because it is impossible to obtain consistent settings by the restriction of already inconsistent settings.

Since inconsistencies are frequently detected before backtracking would reach the FDT leaves, only a part of the FDT is traversed by the search algorithm. We call this part the *decision tree* (DT). The number of nodes in the DT, i.e. the number of traversed

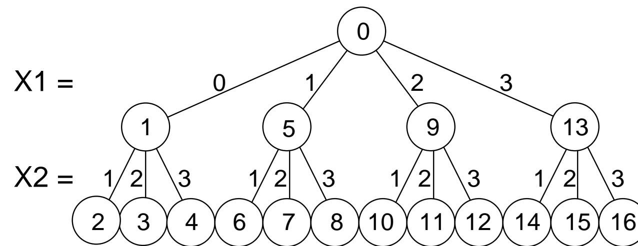

Figure 4.2: Exemplary decision tree

nodes in the FDT, is one of the indicators of the algorithm efficiency. Since most of the time needed by a backtracking algorithm is consumed by exhausting subtrees where no solution exists, the traversed DT number is a good platform-independent measure for algorithm efficiency.

Example

Consider the following CSP: $D_1 = \{0, 1, 2, 3\}$; $D_2 = \{1, 2, 3\}$

$$X_1 + X_2 = 4 \qquad X_1 = 2 * X_2 + 1,$$

and assume that we want to find all solutions. A dumb backtracking algorithm that simply assigns values and checks for consistency only if all the variables of a constraint are assigned would proceed in the node order of Figure 4.2, i.e. it would traverse the entire FDT for finding the only goal node 14. A more sophisticated algorithm which notices that an even value for X_1 cannot satisfy the second constraint would visit only nodes 0, 1, 5, 6, 7, 8, 9, 13, 14, 15, 16, a significantly smaller DT.

The *brute force* search method can be also regarded as an extreme sort of backtracking. Here a decision consists in assigning values to all variables simultaneously, and checking whether or not the assigned tuple is a solution. If the tuple is inconsistent, then the assignment is retracted and another one is selected. In this case, the decision tree consists of only one level, and the expected number of the traversed nodes is extremely high. This obviously very inefficient method calls our attention to the fact that the balance of the decision tree is very important. A large extent of restrictions during one decision can cause the decision tree to swell unnecessarily, which eliminates the opportunity of detecting inconsistencies early, without traversing large subtrees. On the other hand, too little restrictions increase the number of internal nodes with respect to the number of leaves, which makes the expected number of traversed nodes grow.

4.3 Existing CSP solving methods

The field of efficient constraint solving has already a wide literature. In the following, we give a short summary on the existing ideas that can decrease the average time demand of "dumb" backtracking. Unfortunately, most of these techniques have been elaborated and analysed for *binary* CSPs where constraints are defined over at most two variables.

However, a few techniques can be adapted for more complex constraint networks, at the cost that involved data structures become more complex as well.

4.3.1 CSP preprocessing techniques

Constraint preprocessing or *filtering* aims at the better representation of the CSP, i.e. to transform the CSP into an *equivalent* one which has smaller variable domains or more appropriate constraint sets. We call here two constraint networks *equivalent* if they have the same solution vectors.

Consistency algorithms try to exclude some values from the variable domains prior to the solving procedure. Values (or value pairs) are deleted if they cause *local inconsistency* in adjacent binary constraints. *Node consistency checking* considers only unary constraints. If some value of the constraint's only variable does not satisfy the constraint, then it is deleted from the domain. Mackworth [47] has introduced the concept of *arc consistency*: a binary constraint $C(X_i, X_j)$ is arc-consistent if for every value $x \in D_i$ there is a value $y \in D_j$ such that $C(x, y)$. If a constraint is not arc-consistent, there are superfluous values to delete from the domains of the variables. Montanari's *path consistency* algorithm [48] uses the following definition: A *constraint path* is defined as a path in the binary constraint graph. A path of length m through variables X_{i_0}, X_{i_1}, ... X_{i_m} is *path-consistent* if for any value $x \in D_{i_0}$ and $y \in D_{i_m}$ there exists a value sequence $z_1 \in D_{i_1}$, ..., $z_{m-1} \in D_{i_{m-1}}$ such that

$$C_{i_0 i_1}(x, z_1), C_{i_1 i_2}(z_1, z_2), \text{ ..., and } C_{i_{m-1} i_m}(z_{m-1}, y).$$

Verbally, a path-consistent value pair in $D_{i_0} \times D_{i_m}$ should be allowed by every existing constraint path between X_{i_0} and X_{i_m}. If a pair is not path-consistent, the value pair should be prohibited and this information be recorded for the search, e.g. by inserting new constraints into the CSP.

In [50] polynomial algorithms for achieving arc and path consistency have been examined. It has been shown that the time complexity of obtaining arc and path consistency is $O(cd^3)$ and $O(n^3 d^5)$, respectively, where n and c and denotes the number of variables and of binary constraints, while d stands for the cardinality of the domains (assuming equal domain sizes).

4.3.2 Forward schemes

Forward schemes try to optimise *variable selection* or *value selection*.

Freuder provided a variable ordering by static CSP preprocessing [51] and gave sufficient conditions for the search to be backtrack-free. He has shown that arc-consistent trees and path-consistent tree-like structures are *easy* (backtrack-free) CSPs. Dechter and Pearl identified larger classes of easy problems [52]. Still in the area of easy CSPs, Dechter presented the *cycle-cutset decomposition* method [53], which makes early loop-cutting variable instantiations so that the resulting constraint graph remains a set of trees. The cutset-phase finds cutsets as if it were a separate backtracking, and in ev-

ery leaf of this upper layer the efficient tree-solving algorithm is started until a global solution is found.

Look-ahead algorithms [54, 55, 56] avoid the assignment of values that could cause inconsistency in the future. *Partial looking ahead* methods make sure that every variable to assign in the future has at least one value consistent with already assigned variables. *Full looking ahead* algorithms also check that these future variables have values compatible with each other, but the additional overhead is significant. *Forward checking* is a kind of partial looking ahead method with remembering already done consistency checks by means of special variable-value table structures. *Backchecking*, on the other hand, remembers already detected inconsistencies. It makes fewer consistency checks than forward checking, at the cost that it makes more backtrack steps. Gaschnig's *backmarking* algorithm [57, 58] is backchecking with an additional feature. If the algorithm tries to assign again a value to variable v after a backtrack of v, it checks value consistencies only against those variables that may have been changed after the first visit.

Haralick et al. compared these methods on the benchmark of the n-queen problem and random CSPs [59]. He found that after the traditional backtracking, which performs the worst, backchecking, full looking ahead, partial looking ahead, backmarking, and forward checking follow in the order of increasing efficiency.

4.3.3 Backward schemes

Backward schemes control what to do in dead-end situations in order to minimise subsequent tree search.

Backjumping

Backjumping [60, 61] aims at going backward several levels in the DT when backtrack follows, instead of going back to the most recent node in the tree, thus skipping possibly large FDT subtrees without the risk of losing solutions. The backjump target level can be determined by exploiting topological properties of the constraint network. If there is no consistent value available at the assignment of variable v, the backjump target must be one of the variables adjacent to v in the constraint graph. To avoid losing solutions, the most recently assigned of these neighbours is selected for the new assignment, and all decisions made after the assignment of this target are withdrawn.

Backjump target calculation requires the maintenance of a list TLIST of possible targets, a global variable. In the following code, $P(X_i)$ is the set of variables that precede X_i in the variable ordering and are adjacent to X_i through a constraint.

BackjumpTarget(X_i)
  if X_i is the first variable in the ordering **then**
    exit. No solution exists.

$TLIST \leftarrow TLIST \cup P(X_i)$
if $TLIST = \emptyset$ **then**
exit. No solution exists.
$X_j \leftarrow$ the largest indexed variable in $TLIST$
$TLIST := TLIST \setminus X_j$
return j

It should be emphasised that only in dead-end situations is the backjump target the most recently assigned neighbour. When a backjump is immediately followed by another backjump, the candidate list (TLIST) may include other variables as well.

Learning

In CSP context, *learning* means transforming some recorded information into explicit constraints. When we attempt to assign a value to X_i and recognise that there is no compatible value for it (a dead-end situation), it is obvious that the already assigned tuple $X_1 = x_1$, ..., $X_{i-1} = x_{i-1}$ is inconsistent and no solution will incorporate this tuple. We call this tuple a *conflict set.* We could add a constraint over the first $i-1$ variables that excludes the conflict set to prevent it from reoccurring. However, there is no point in doing that, because backtracking will never reassign the conflict set. On the other hand, if the conflict set contains a smaller conflict set that disallows every value of X_i, it may be worth recording the smaller set as an explicit constraint, because it might reoccur later.

The task is therefore finding subsets of the original conflict sets that are conflict sets themselves. Obtaining smaller sets by excluding *irrelevant* variable-value pairs constitutes *shallow learning.* A pair $X_j = x_j \quad (j < i)$ is said to be irrelevant with respect to X_i if it is consistent with every value of X_i. (Note that we are still discussing binary CSPs.) *Deep learning* consists in identifying *minimal conflict sets* [62], such sets that contain no conflict subsets. Obtaining all minimal conflict sets is usually quite a time-consuming procedure, may cause severe space complexity problems, and there is no guarantee that the added constraint will not slow down the backtracking procedure. It is therefore usual to limit the search for minimal conflict sets in terms of constraint number, constraint size, and required time. Dechter gives a comparison of the performance of various types of learning techniques [53].

4.4 Constraint-based ATPG modelling in CONTEST

We present Tilly's CONTEST ATPG tool [25] in details so that existing gate-level constraint-based ATPG techniques can be identified.

4.4.1 ATPG problem representation in CONTEST

The constraint network is constructed the following way from the ATPG problem:

- CSP variables represent circuit signals. Since ATPG is a composite simulation of the fault-free and faulty behaviours, each variable has a domain $D_i = \{0, 1, D, \overline{D}\}$, as described in Section 3.2.2.
- The test requirements are expressed by confinements in the *initial domain* of the variables:
  - The faulted variable is set to {D} or {D̄}. (CONTEST uses the single stuck-at fault model.)
  - The output where the discrepancy is expected is confined to the domain {D,D̄}.
  - The domain of the variables that can be influenced by the fault is the full set {0,1,D,D̄}.
  - The domain of the variables that cannot be influenced by the fault is limited to {0,1}.
  - *Irrelevant variables*, i.e. those not in the *coverage cone* of the selected output, are removed from the network along with the constraints defined over them.
- Each gate is represented by a constraint. A constraint is internally represented with a so-called *Dynamic Binary Constraint Network* (DBCN). A DBCN generates the locally consistent tuples by a *DBCN interpreter*. The main advantage of using DBCN is that it requires less space even for 3-4 input gates than *truth-table based* descriptions. The constraint representing the gate that drives the fault site is modified somewhat.

The variable assignment x_{CSP} in the CSP is transformed into a bit vector in the following way:

1. Disregard the values of the variables that do not correspond to an input pin. Let the resulting vector be $x_{ATPG} = \{x_1, \ldots, x_n\}$ (n is the input number).
2. If $x_{ATPG_i} = D \quad (1 \leq i \leq n)$ then $x_i \leftarrow 1$. If $x_{ATPG_i} = \overline{D}$ then $x_i \leftarrow 0$. These cases can only happen when an input signal is stuck-at.

x_{ATPG} is a test vector if x_{CSP} is a CSP solution.

circuit	time (sec)		fault coverage (%)	
	backtracking	backjumping	backtracking	backjumping
c432	2258	470	90.81	99.23
c499	14371	97	53.73	98.94
c880	142	18	99.77	100
c1355	50987	2207	24.95	97.7
c1908	58432	1826	53.28	60.3
c2670	57325	5989	60.3	85.1
c3540	72541	20318	35	81.5
c5315	73208	8548	76.3	93.14
c6288	54221	44469	62.27	67.18
c7552	121239	19338	47.28	86.34

Table 4.1: CONTEST results

4.4.2 CSP solving in CONTEST

CONTEST has a dedicated built-in CSP solver. The decision engine of CONTEST differs from "conventional" backtracking (Section 4.2.1) in the sense that all variables of a gate constraint are assigned simultaneously during a decision. The gates that are candidates for decision selection are placed on the *hesitation queue*, a queue on which forward heuristics operate.

CONTEST extensively uses *implication* (Section 3.2.3). The multiple-valued domain of the variables has a maximal power to express value exclusions made by the implication procedure. Since 16 actual domains exist for 4 values (0,1,D,$\bar{D}$), the implication part of CONTEST resembles to the 16-valued logic of Hegedüs [21], but the gates are described more briefly.

Dead-end situations are resolved by a *contradiction resolution algorithm* which is based on the principles of *backjumping* (Section 4.3.3). Since the original backjumping algorithm assumes single variable assignments during decisions, value dependencies are maintained by more complex data structures.

The performance of the CONTEST CSP solver has been evaluated on the standard gate-level ISCAS benchmarks (Appendix A, [11]) using a SUN-10 workstation. Table 4.1 shows the performance figures of CONTEST.

Chapter 5

ATPG modelling in BudaTest

This chapter describes the constraint-based ATPG modelling technique of the BudaTest architectural tool. As constraint-based ATPG modelling already exists at the gate level (Section 4.4), I will underline the new concepts that solve problems arising at the architectural level. The source of these problems can be one of the following:

- The large domain of data signals and the high-level component description makes traditional variable and constraint representation very inefficient.
- The architectural description, a CAD tool output, contains such elements that do not appear at the gate level. Such problems include wide signals, hierarchical descriptions, wiring of half-words to components, and fault modelling issues.

Since BudaTest is intended to operate at the *architectural level* (Section 2.2.3), it accepts a circuit description conforming to that style as input. The input language is the structural subset of the IEEE standard VHDL [10].[1]

5.1 The constraint network

5.1.1 Variable representation

We define two constraint variables for each signal appearing in the architectural description. Both variables have the same domain, equivalent to the type range of the signal. The variables represent the signal's value in the fault-free and faulty simulation.

This style is contrary to gate-level signal modelling, which handles together the two signals (see the meaning of the 0, 1, D, and $\bar{D}$ values in Section 3.2.2). However, joint handling would involve a quadratic domain size, which is prohibitively large at the architectural level.

In addition, bookkeeping of what values are actually permitted from a domain requires also prohibitively large space. For instance, to store the actual domain of a 16-bit

[1] BudaTest contains an ISCAS parser too.

integer would require 2^{16} bits. Although variable storage is not a crucial problem in current environments, the saving/restoring operations with these values would slow down backtracking.

5.1.2 Constraint representation

In gate-level ATPG tools the small number of consistent tuples allows for the table-based enumeration of component functions. This method, however, is deficient again at the high level. The large width of data signals immediately leads to immense storage complexity even at the description of a simple arithmetic or auxiliary unit. For example, a 2-to-1 multiplexer operating on 8-bit integer numbers has 2^{17} different ways to set the inputs, and a table describing the constraint would contain so many rows.

For this reason, while admitting that enumeration is an efficient technique wherever the small number of tuples permits its use, we must provide an alternative way if we want to move forward high-level complex components. The consistent value sets of the 2-to-1 multiplexer component could be briefly described with a simple *rule*: *When the selector input holds value 0, the output value should equal* input 1*, otherwise it should equal* input 2.

This implicit description defines the same consistent tuple set as the enumeration-based one, but the consistent tuples cannot be immediately read out of it. Instead, a *function* that uses the implicit definition for the consistency check must be provided. This is easily implemented in the BudaTest object-oriented library-based ATPG system.

5.1.3 CSP-based ATPG problem formulation

The TPG problem poses the following question: *Considering a given physical fault, which circuit input vector produces different responses during the fault-free and faulty operation of the circuit?*

The answer for this question can be determined by means of *fault simulation.* During that, the operation of two circuits (the fault-free and that containing the target fault) is simulated together for various input vectors.

The verbal requirements against a test vector are easily formulated as a CSP. The constraint network representing the TPG problem is built up the following way:

- Two variables are defined for each circuit signal as described in Section 5.1.1. The variables representing the fault-free simulation are *fault-free domain variables* while the remaining ones are *faulty-domain variables.*

- The constraints of the network are derived from the circuit functionality, the fault effect, and the observability condition:

  1. The majority of constraints are derived from the circuit components. Two identical constraints are generated from one component: the first is defined

over the fault-free domain variables that represent the signals the component is connected to while the second is defined over the equivalent faulty-domain variables. These constraints restrict value combinations so that they represent the component functionality, like in the case of the multiplexer example above.

2. The presence of the target fault is a local perturbation of the faulty domain in the constraint network. The kind of modification depends on the exact fault type, and will be discussed in details later in Section 5.6. As a frequent case, a stuck-at fault of a single-bit signal is represented by the duplication of the faulty-domain variable and the addition of a *unary* constraint which permits the duplicated instance to take only the stuck-at value.
3. The fault effect observability prescribes that at least one output signal take discrepant values in the two domains. If there is only one circuit output, this condition is represented by an inequality constraint between the two output variables. Otherwise, a somewhat more complex constraint is defined over all outputs variables.
4. The input variables must take the same values in the two domains. This is expressed by equality constraints.

Figure 5.1 depicts a small exemplary circuit with the ATPG-purpose constraint network derived from it. The constraints C_1, C_2, and C_3 are derived from the components of the circuit. They are present in two instances, defined over the faulty-free domain and faulty domain variables. The unary constraint *fault*, permitting only the stuck-at value, is responsible for implementing the effect of the stuck-at fault. The inequality constraint $\neq$ assures that the effect of the fault is observable on the output.

The following theorem proves the *equivalence* of the ATPG problem and the obtained constraint satisfaction problem:

Theorem 1 *If a value assignment is a CSP solution, then the values of the fault-free domain input variables are a test pattern for the target fault. Every test pattern is part of a CSP solution.*

Proof: Both statements follow from the way the CSP is constructed. Assume that x_{CSP} is a CSP solution. Let x_{ATPG} be the vector consisting of the fault-free domain input variable values. Since the constraints of Type 1 define exactly one output value for every input combination, they describe what functions the components implement. Thus the values of the fault-free and faulty domain variables (except the auxiliary variable inserted into the faulty domain) describe what values the signals of the fault-free and faulty circuits take when simulated without and with the presence of the fault. The fault perturbation constraint assures that the faulty domain variables are influenced by the fault value the same way as the faulty simulation is influenced by the injected fault. The observability constraint assures that at least one output is discrepant, so x_{ATPG} is indeed a test pattern.

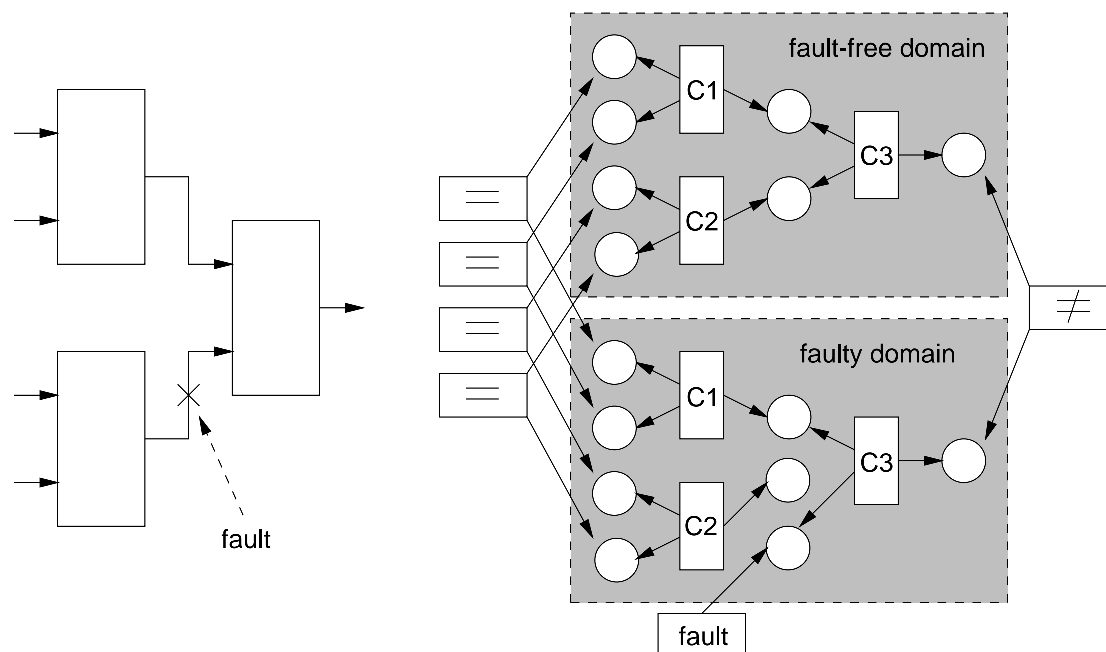

Figure 5.1: ATPG problem represented as a constraint network

Now consider a test pattern x_{ATPG}. Simulate the behaviour of the circuit for this pattern and assign the obtained values to the CSP variables. We show that all constraints are satisfied:

- Component-derived constraints are satisfied because of their construction method.
- The fault effect constraint is satisfied because a fault activated by x_{ATPG} produces exactly the value allowed by the unary *fault* constraint.
- The observability constraint is satisfied otherwise x_{ATPG} could not be a test pattern. □

The proof can be obtained in a similar way for fault classes other than the single stuck-at one (see Section 5.6).

5.2 Advantages of the constraint technique

The constraint technique has been chosen to be the ATPG platform on account of several advantageous features.

- Constraints are a very generic descriptive tool with a high modelling power. In the constraint-based ATPG representation, *everything* is modelled as a constraint. All test requirements can be handled in a *uniform* way, therefore the constraint solver algorithm does not have to distinguish between irregular cases. The circuit

architecture, the fault effect and additional test conditions constitute together a uniform constraint network.

- In addition to uniformity, constraints are an *open* representation; new concepts and ideas are as easily added to the ATPG system as they can represented by constraints, which is usually the case.

- Constraint satisfaction problems and constraint solving algorithms already have a wide literature. Unfortunately, a large part of the literature is elaborated and analysed for *binary* CSPs (Section 4.1). For this reason and for the particularity of the ATPG problem, this dissertation proposes an ATPG-specific constraint solving method, which still includes many general constraint solving ideas and accelerating techniques. For example, such techniques known from the literature as *implication*, *domain filtering*, and *assignment heuristics* can be employed with adequately chosen data structures.

- One of the important characteristics of constraints is their *symmetry*. In contrast to a model with output signals given as a function of input signals, constraint have *related variables* without a predefined direction of data flow. During constraint solving the values are frequently assigned earlier to internal variables than to inputs. Symmetric constraints facilitate *backward implication* (data propagation against the data flow direction of the original circuit) due to their symmetric representation (Figure 5.2).

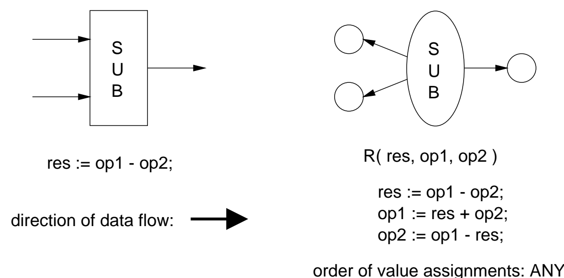

Figure 5.2: Symmetric constraints

- Since every kind of the requirements can be defined by uniform constraints, a non-central, heuristics-controlled solving algorithm can be used. In such an algorithm, there is no prescribed order for decision types, as opposed to many gate-level algorithms, including D and PODEM (Section 3.2). Instead, more freedom is left for cost-function based heuristics which can control the search according to the circuit characteristics.

5.3 Hierarchical support

Besides efficiency, support for state-of-the-art design methodologies has a priority in this work. Since many levels of hierarchy are introduced during the design process (see Section 2.2), the compliance with the hierarchical design methodology is essential in the ATPG methodology. This is achieved by support for *hierarchical circuit modelling* and *hierarchical fault modelling* (Section 3.3.2).

5.3.1 Hierarchical options in BudaTest

The constraint library of BudaTest contains the rule-based (high-level) arithmetic descriptions of the utilised components. As argued in Section 3.3.2, the exact representation of component faults necessitates an alternative, low-level representation of these components. BudaTest provides several options for the low-level representation:

1. *structural flattening.* As the examined component can be replaced with a set of interconnected subcomponents, the faulty-domain constraint representing the component can be replaced with a smaller constraint network reflecting the internal structure of the component (Figure 5.3). Obviously, the solution of the extended CSP will be a test vector for the top-level circuit. The replacement can be done in accordance with the recursive technique shown in Figure 3.5 in order to have minimal parts represented at the low level.

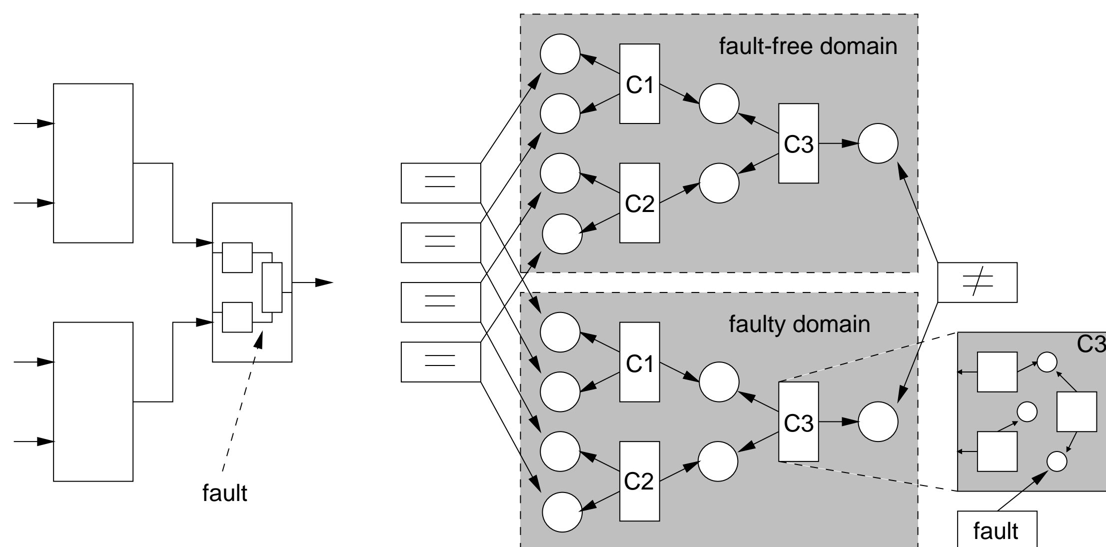

Figure 5.3: Network flattening

2. BudaTest can be interfaced with low-level ATPG tools. The tool calculates test for a component, then BudaTest *expands* the generated component test to the top level of the hierarchy using the high-level constraints. The tool should be able to

generate consecutive patterns in case contradictions occur during the high-level expansion.[2]

3. *component test description files* (CTDF). Hierarchical testing can be simplified in a library-based approach like BudaTest. After the ATPG for a given component is accomplished, the test patterns are stored as a CTDF in the library together with the high-level constraint representation. If the library element is used within a higher-level architecture, either the high level constraint equivalent or the CTDF is used, depending on whether the target fault is located outside or inside the given component. In the latter case, the high-level ATPG task becomes again *test expansion*. The advantage of this method is the *reuse* of already computed test patterns. It is also advantageous because only one level of abstraction is used at a time.

5.3.2 Problems in CTDF expansion

A CTDF-based ATPG procedure may face space complexity problems. A component test expansion can fail at the top level, because test patterns can not be necessarily propagated to the component inputs and the component output is not necessarily observable at the top level. To avoid the risk of losing solutions, the only possibility is that the CTDF contains *all* test patterns of a component. In contrast to other algorithms or to BudaTest without the CTDF extension, the storage requirement of the technique is not linear, because it must store every component test pattern along with the expected and faulty-domain results in the CTDF. *Don't care* outputs must be also determined, because the may affect the propagation of the component output discrepancy. These requirements may inflate the size of a CTDF, which grows rapidly if we move up the hierarchy. Nevertheless, when the CTDF size becomes intolerable, it is still possible to fall back to the *structural flattening* technique.

Another trade-off is to limit the size of a CTDF. This involves the risk of CSP solution loss, so we must be aware of the fact that the fault coverage of the methodology will decrease. Table 5.1 reports the circuit-level coverage when the CTDF size is limited to a single test line per fault. The benchmark set is described in Appendix A.

The flattening technique involves no solution losing, therefore the corresponding column contains the ratio of detectable faults. The column of *top CTDF coverage* indicates the coverage when top-level components are replaced with their limited CTDF. The decrease in fault coverage is usually 5-10%. In the circuits of more than two levels of hierarchy, the cost of using 1-line limited CTDF is about the same each level. We note that having a higher limit for CTDFs would bring much better results.

[2] If the utilised component tool is BudaTest itself executed on a low-level constraint network, we obtain the case of *structural flattening*, where decisions are made within the examined component first.

circuit	levels of hierarchy	flattened coverage	top CTDF coverage	total CTDF coverage
adder3	2	93.02%	87.07%	87.07%
adder5	2	95.77%	92.18%	92.18%
adder6	2	96.47%	93.47%	93.47%
adder7	2	96.96%	94.40%	94.40%
adder8	2	97.34%	95.09%	95.09%
multiplier 2x2	3	72.34%	67.71%	60.56%
multiplier 4x4	3	80.00%	62.21%	58.69%

Table 5.1: Effect of limited CTDF size on the fault coverage

5.4 Control-dominated and sequential circuits

5.4.1 Control-dominated circuits

We call a circuit *control-dominated* if its architectural description contains many *control signals* between the *data part* and the *control part* (see Section 2.2.3 for the definition of these terms).

BudaTest poses no restrictions on the number of such signals. Since the value of control signals can be usually used very well in *rule-based component descriptions* (Section 5.1.2), the existence of such signals involves no problems in modelling. The token propagation technique of BudaTest (Section 6.4) makes extensive use of the control signals.

On the other hand, we expect that the *control part* component, responsible for processing the Boolean result signals and generating other control signals, can be implemented with a few (1-2 dozens) logic gates. The reason of this limitation is that the control part component, different for each design, cannot be part of the constraint library. It is inserted into the constraint network as a number of synthesized logic gates which process control input as it were simple data.

5.4.2 Sequential modelling

At the gate-level a sequential circuit is defined as one containing combinational logic, a feedback loop and a register array (Section 3.2.4).

Feedback sequentiality

BudaTest supports sequentiality by the *iterative array* model. As discussed in Section 3.2.4, using this technique multiplies the number of variables and therefore inflates the state space of the problem. Searching for a one frame longer test sequence implies the creation of a new instance of every variable which multiplies the state space by its

domain. Thus the iterative array representation is mainly applicable for data-dominated circuits where most of the values in every time frame indeed matter. For circuits where many faults require several clock cycle long state transition sequences but the data values are unimportant this method is impractical because of the extensive but needless data computations. In [44] a promising ATPG approach for such circuits is presented, which deals with explicit state transitions and not with the time frame concept.

Note, however, that the type-uninterpreted *colouring technique* of BudaTest (described in Section 6.4) is aimed at the elimination of redundant data variables. This technique efficiently reduces the overhead when BudaTest is executed for moderately sequential circuits, as the experiments performed on sequential benchmarks show.

The expected test length is an important question, because this determines the size of the CSP. In the BudaTest system *frame incrementation* is used to determine this length. First a one time frame-long sequence – a vector – is sought. In case of failure, the expected length is increased before every new search one by one. This is repeated until a test is found or the frame limit is exceeded. As a result, the shortest possible test sequence is found. The first searches may turn out to be superfluous if there is a length minimum for the test sequence, but they are still worth trying, because the wasted time is only a fragment of the time consumption of the next run. In addition, the time required for the last run is reduced, because there is no point in expecting discrepant responses on the outputs belonging to the previous frames.

BudaTest assumes the same circuit restrictions as described in Section 3.2.4, with one difference. Registers do not need to have *reset* (though having *reset* is more efficient), because the *unknown* logic used in the *colouring technique* can handle this case.

Component sequentiality

In hierarchical testing the use of sequential components is a new source for sequentiality. When such a component is unrolled by the iterative array technique, the constraint model of the component would not be a relation over the variables, because its evaluation would depend on its actual *state*.

BudaTest supports sequential components only if the state registers appear explicitly in the architecture. This can be regarded as *flattening* the component in the fault-free domain as well, but the *output logic* and the *next-state logic* of the component can be modelled at the high level.

5.5 Array selection

Accessing selected elements of arrays is a fundamental feature in hardware descriptions. For instance, `vectorsignal( index )` selects an individual element of an array, while `vectorsignal( lowbound to highbound )` refers to a part of a signal.[3] It is common

[3]These are called *indexed* and *sliced expressions* in VHDL.

in hardware architecture that, for example, a half word is used by some component, and the other half goes to the parity checker. However, in traditional CSPs the term of referencing constraint variables partially is undefined.

In gate-level ATPG approaches this problem does not occur, because bit arrays are treated as sets of separate bits and components are wired to bit-level signals. The most essential feature of architectural ATPG is, however, the joint handling of bits of wide data variables, which involves complex arithmetic operation over abstract variables. Constraints defined over partial variables should be permitted.

The CSP derived by BudaTest from a VHDL description supports this feature. We obtain the desired modelling effect by the duplication of the referenced variable and the introduction of a new constraint. The original variable has the length and domain of the referenced array signal, while the domain of the added one corresponds to the array selection. The *mask* data structure (Section 6.2.2) permits efficient data propagation between the two variables. Of course, since the selected array part is not a separate physical signal, faults are not injected into it.

5.6 Fault model

Fault modelling is a crucial issue in every ATPG technique. There is no point in inventing ultra-fast algorithms if what the generated test detects is not the effect of realistic physical faults. It must be therefore shown that the employed constraint modelling methodology is capable of modelling the effect of physical faults.

Since constraints are a highly generic modelling tool, the fault model is as appropriate as relevant faults can be represented by constraints. In the following part, individual fault classes will be investigated and shown to be representable by constraints.

The coverage of the *gate-level stuck-at* and *short model* (Section 3.2.1) will be proven. The stuck-at fault model is commonly acknowledged by the test community to sufficiently represent the effect of physical faults. Short faults are used less frequently.

Theorem 2 *All gate-level stuck-at faults of a synthesized architectural circuit can be represented by the introduced constraint modelling technique and hierarchical modelling.*

Proof: We deal first with top-level interconnections. The constraint representation of stuck-at faults affecting single-bit signals is really straightforward. The fault effect is simply injected into the network by the duplication of the faulty-domain variable representing the faulty signal. The constraint representing the component which drives that signal will be defined over the new variable instead of the original one. In addition, a unary constraint which permits only the stuck-at value is connected to the original variable, as shown in Figure 5.1.

If a single bit of a wide signal is stuck, a binary constraint is inserted instead of the unary one (Figure 5.4). The constraint enforces equality on the unaffected bit positions,

and permits only the stuck-at value at the fault position of the original variable.[4]

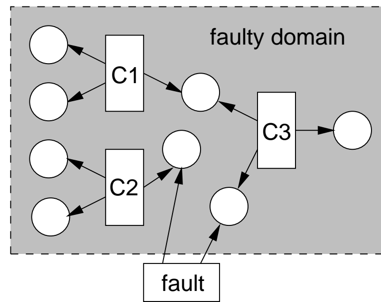

Figure 5.4: Representation of stuck-at faults of wide signals

The exact modelling of component stuck-at faults is performed in the hierarchical way as described in Section 5.3. *Structural flattening* will reach sooner or later the level where the gate-level signal appears as an explicit interconnection. We can perform here the described constraint network perturbation.[5] □

The modelling of short faults requires the duplication of the faulty-domain variables of the two shorted signals. Then, a constraint defined over the four variables is inserted into the network. This constraint prevents the duplicated variables to take different values, and the common value is determined by the values of the original pair. The allowed value settings are listed in Table 5.2. This table applies for wired-and short faults. Values should be negated to represent a wired-or short fault.

original variable 1	original variable 2	duplicated variable 1	duplicated variable 2	comment
1	1	1	1	inactive fault
0	0	0	0	inactive fault
1	0	0	0	active fault
0	1	0	0	active fault

Table 5.2: A constraint representing a wired-and short fault

Short faults must not cause feedback and introduce new states in the faulty behaviour. This is a common expectation in every ATPG algorithm. Due to the large number of signal pairs in a circuit, tests for short faults are worth computing only if the fault is relatively likely to occur.

Though the discussed fault classes are considered the most standard ones, the constraint technique allows for the inclusion of many other fault types. For example, for ATPG program debugging purposes a hypothetical fault type, *full-width stuck at abstract*

[4]Note that different constraints are needed for different data encoding styles.

[5]We note that in sequential circuits the implicit *clock* and *reset* signals do not appear in the constraint model.

value, was also widely used.

Constraint modelling is not limited to single faults. Since there is no fault-dependent part of the ATPG algorithm (a general CSP solver), as many perturbations can be made in the faulty domain as desired.

Chapter 6

CSP solution

This chapter describes what techniques are used for obtaining the solution of the constraint satisfaction problem. We discuss here which existing ATPG and CSP solving methods can be exploited in the high-level constraint-based ATPG. Our most important goal is to highlight acceleration possibilities inherent in the high-level representation of the circuits.

We describe the constraint solving engine of the BudaTest ATPG tool, developed by the author under the FUTEG (Functional Test Generation and Diagnosis) project [2]. The following goals motivated the development of BudaTest:

- *to exploit existing CSP solving ideas.* I wish to exploit existing generic CSP acceleration techniques described in Section 4. Since most of the general CSP solving algorithms have been developed for *binary CSPs* and analysed almost exclusively in binary CSP environments, the described ideas had to be tailored to non-binary CSPs where constraints are defined over an arbitrary number of variables (see Section 5.1.3). In some cases, this means only a somewhat more complex data structure behind the same algorithm. In other cases, an entirely different algorithm was developed which considers the same principles.

- *to exploit that the CSP is a special ATPG-dedicated one.* I would like to show that a solving engine that exploits acceleration opportunities can perform much better than general CSP solving methods.

- *to exploit the features of the high-level circuit description.* I would like to show that if we exploit high-level information inherent in architectural circuit descriptions, we can efficiently moderate the time consumption required by exponentially growing state spaces.

6.1 Backtracking in BudaTest

As every constraint solver, BudaTest uses *backtracking* (Section 4.2.1) to obtain the CSP solution. Of course, the basic backtracking algorithm is enhanced with various features

borrowed from the literature or developed by exploiting architectural characteristics.

A *decision* or a *forward step* is an assignment of a value to a constraint variable. After each decision, an *implication* step is performed to reveal implied effects of the decision.

A *backward step* is the withdrawal of a previous decision, including the restoration of the previous constraint network state. *Backjumping* (Section 4.3.3) is used in BudaTest so that many backward steps can be performed at the same time without losing CSP solutions.

6.2 Forward techniques

6.2.1 Implication

Implication means exploring the implied equivalent restrictions of a decision. Some form of implication has been incorporated in every ATPG algorithm due to the reduced resulting problem subspace (Section 3.2.3).

We will use the following terms in the discussion of implication:

- The *actual domain* D_{Ai} of a variable X_i are those that are regarded by the CSP solver as possible elements of a solution. To represent actual domain, the algorithm needs to have some dedicated data structure.
- A variable X_i is *bound* when its actual domain consists of one value x_i. Otherwise, the variable is unbound.

A decision is an assignment of a value x_i to a variable X_i. After that, the constraints defined over X_i are called to perform a *consistency check*. Let $X_B = \{X_{B_1}, \ldots, X_{B_k}\}$ denote the set of bound variables of the constraint ($X_i \in X_B$), and $X_U = \{X_{U_1}, \ldots, X_{U_l}\}$ the set of still unbound ones. A traditional consistency would examine whether there exist values in the actual domain of every unbound variable so that they are related with the already assigned tuple. Formally, the consistency check of constraint C passes if

$$\forall i \quad (1 \le i \le l): \quad \exists x_{U_i} \in D_{AU_i}: \quad C(x_{B_1}, \ldots, x_{B_k}, x_{U_1}, \ldots, x_{U_i}, \ldots, x_{U_l}).$$

The decision on X_i is approved if all the constraints defined over X_i pass the consistency check. Otherwise, it is a contradiction and a backward step follows.

Implication not only checks consistencies but tries to exclude values from the actual domain of the adjacent variables. If there is a variable value for which there exists no related assignment to the other unbound variables then the value is excluded from the domain:

$$\forall i \quad (1 \le i \le l): \quad \exists x_{U_i} \in D_{AU_i}: \quad \forall x_{U_j} \in D_{AU_j} \quad (1 \le j \le l, i \ne j):$$
$$\neg C(x_{B_1}, \ldots, x_{B_k}, x_{U_1}, \ldots, x_{U_i}, \ldots, x_{U_l})) \Rightarrow D_{AU_i} \leftarrow D_{AU_i} \setminus x_{U_i}$$

When an adjacent variable is restricted, the implied restriction is part of the decision. Since the restricted variable may participate in other constraints, they are also called to perform implication. The implication procedure is therefore an iterative sequence of primitive implication steps performed by the constraints. The constraints waiting for implication are added to an *implication queue*. As implication means always a restriction in some actual domain, the queue will sooner or later become empty and implication terminates.

The practical implementation of implication is a different question. Two aspects must be taken into account:

1. The representation of actual domains requires data structure support. The next section deals with this issue.

2. It would be very impractical to organise deeply nested loops in order to explore *all* value exclusion possibilities. Implication must be balanced to prevent the time required by implication from impairing the gain of value exclusions.

The implication functions of the BudaTest constraint library elements are written in the following way:

1. When all the variables of the constraint are bound, it performs a consistency check. This itself is sufficient to obtain a correct CSP solution.

2. If there are unbound variables, the function tries to extend the effect of the activating decision. Of course, inconsistencies can be detected in this phase as well. The effect propagation is not performed by the enumeration of all possible value combinations of unbound variables (as described above), but by means of *rule-based* functions. The implication primitive is therefore not a fixed procedure executed on different tables (as done in gate-level algorithms), but a different function varying with the modelled architectural-level components (a *virtual* C++ function).

   A 2-to-1 multiplexer constraint, for example, can examine if the *selector* variable is assigned. If it is, then the constraint becomes an equivalence between the selected input and the output. If no, we may try to apply other rules (e.g. check if the output cannot match one of the inputs)), or leave the function thinking that the expected gain is not worth the extra check.

   Note that the result of the backtracking search does not depend on this implicative behaviour, but its performance does. In the extreme hypothetical case of having every constraint with empty propagation body we simply fall back to normal backtracking when consistency is verified after the variables are set.

As the presented multiplexer example shows, the efforts made to perform implication is the choice of the developer of the constraint library. Gathering *all* information implied

by a decision would be as time-consuming as the backtracking procedure itself. In fact, a failing search in a decision subtree can be regarded as a perfect implication detecting the inconsistency, performed only in the subtree node.

Employing implication involves some modification in the original backtracking program. We must keep a record of all variable restrictions for each DT level, used for the restoration of previous search states in case backward steps are needed. (Without implication, this is a single change each DT level.)

Implication, since it performs significant reduction in the remaining subspace after every decision, is a very powerful enhancement. It is easy to show that it covers entirely *partial looking ahead* (Section 4.3.2) when applied in binary CSPs.

6.2.2 Interval, masked and set logic

Many constraint solvers and ATPG algorithms are equipped with some sort of implication procedure. Since the goal of implication is always the exclusion of values from the domain of certain variables, a suitable data structure supporting exclusion must be chosen. In gate-level ATPG approaches *set* structures are used to represent still permitted values. The *multiple-valued variable domains* of Tilly's CONTEST (Section 4.4, [25]) or the 16-valued logic of [21] can be also regarded as sets. The results of these approaches show that the gain consisting in efficient exclusion is worth the cost of extra administration.

Architectural circuit modelling includes abstract components and signals of abstract data types (Section 2.2.3). Abstract variables (integers, long vectors) are usually difficult to handle efficiently for the CSP solver, because their large domain inhibits the enumeration of still possible values or other kinds of set structures. A variable representing a 16-bit integer signal would consume 8 Kilobytes of memory as a *set* structure, which is prohibitively much for saving/restoring.

The BudaTest approach still wants to make certain forms of data exclusions available. A not entirely unrestricted variable may have *bound*, *interval* or *mask* values. *Sets* are also permitted for signals of moderate domains.

- *bound.* A specific value is assigned to the variable, either as a result of a decision or that implications make it bound. This structure is denoted as B_v where v is the specific value.

- *interval.* The permitted values of a variable are expressed by a lower and a higher bound $[l, h]$. The structure denoted as I_{lh}. Since all the variables have *discrete domains*, there exists a full ordering over the domain values, thus defining intervals is meaningful.

  This *interval* representation has been introduced on account of the frequent use of arithmetic units performing addition and subtraction in the data part, and especially of integer comparisons in expression evaluations (see Section 2.2.3). The

implication procedures of these constraints can be written in a way that the implied restrictions can be expressed in terms of intervals.

- *mask*. The permitted values of a bit array are expressed in terms of assigned and unassigned bit positions. A mask structure is denoted as M_{vm} where v is a specific value and m is a bit mask showing which positions are important.

  The *mask* logic comes again from the features of digital circuits in practical use: component wiring refers very often to array parts only instead of full arrays (Section 5.5). The auxiliary constraints that handle these cases and the injected faults are capable of *mask* data propagation.

- *set*. This data structure is the more general, since all the above described structures could be described by extreme *set* structures. A *set* structure is denoted as S_S where the S index specifies the set of elements.

  For reasons discussed above, a *set* it is very expensive to use, therefore it should be applied for variables of moderate domain sizes where every excluded value really matters, i.e. especially for control-related variables. Using *set* structure matches very well the *colouring* topological processing technique described in Section 6.4.

During implication the constraint functions assign values expressed by these structures to the variables. The new values of the variables are determined by the *intersection* of their actual values with the recently assigned one. The *intersection operator* ($\otimes$) is defined according to Table 6.1.[1] The operation assigns the intersection result to the variable or indicates contradiction if the consistency condition is not met. (If the result is a one-long *interval*, a one-element *set* or an entirely important vector, it is changed to *bound*). The result depends on whether or not the *set* structure is allowed for the given variable. If yes, intersection is always commutative. If no, the variable retains its original value in the indicated cases. Though we do not perform the restriction, this will not lead to false solutions, because implication does not *have to* effectuate the implied changes, just improves efficiency in doing that.

The advantage of the used logic is the following:

- *bound, interval* and *mask* structures require at most twice the variable word size.

- The intersection between these structures can be performed very quickly, using only a few instructions.

- The result can be very often represented with this logic, and no harm happens if not.

[1] $+, \wedge, \vee$ denote bit-wise xor, and, or operations.

operands	set allowed	result	consistent	commutative
$B_v \otimes B_w$		B_v	$v = w$	yes
$B_v \otimes I_{lh}$		B_v	$l \leq v \leq h$	yes
$B_v \otimes M_{wm}$		B_v	$(v + w) \wedge m = 0$	yes
$B_v \otimes S_S$		B_v	$v \in S$	yes
$I_{lh} \otimes I_{jk}$		$I_{max(l,j)min(h,k)}$	$max(l, j) \leq min(h, k)$	yes
$I_{lh} \otimes M_{vm}$	yes	$S_{I_{lh} \cap M_{vm}}$	$I_{lh} \cap M_{vm} \neq \emptyset$	yes
$I_{lh} \otimes M_{vm}$	no	I_{lh}	yes	no
$I_{lh} \otimes S_S$	yes	$S_{I_{lh} \cap S}$	$I_{lh} \cap S \neq \emptyset$	yes
$I_{lh} \otimes S_S$	no	I_{lh}	yes	no
$M_{vm} \otimes I_{lh}$	no	M_{wm}	yes	no
$M_{vm} \otimes M_{wn}$		$M_{(v \wedge m) \vee (w \wedge n), m \vee n}$	$(v + w) \wedge m \wedge n = 0$	yes
$M_{vm} \otimes S_S$	yes	$S_{M_{vm} \cap S}$	$M_{vm} \cap S \neq \emptyset$	yes
$M_{vm} \otimes S_S$	no	M_{vm}	yes	no
$S_S \otimes S_T$		$S_{S \cap T}$	$S \cap T \neq \emptyset$	yes

Table 6.1: Intersection results

6.3 Backjumping

BudaTest implements backward steps by *backjumping* (see Section 4.3.3). Note that the practical implementation of dependency recording is much more complex in an implication-equipped CSP solver than in the traditional consistency-check based binary ones. When we use implication, *assigned* variables, *restricted* variables and *checked* variables are three different things, so the procedure using a simple *parent* list is deficient at backjumping target calculation.

The backjumping algorithm is implemented in BudaTest as follows.

- The list of restrictions (the variable assignment and those made during the subsequent implication) are recorded for every level of the DT. Such a list is created in every forward step and destroyed in every backward step. This is a necessity without regard to backjumping, because it serves for the restoration of previous states as well.

- A list of accessed variables is also maintained for every level of the DT. In forward steps a new *access list* is created. It is not destroyed, however, in backward steps, but it is merged with the *access list* of the previous DT level because a subtree search can be regarded as an implication in the subtree root node, and therefore the accessed variables belong to the dependencies of the subtree root decision.

- At backjump target calculation, the *access list* of the current level is checked against the *change lists* of the previous levels, starting from the most recent one,

and the level of the first hit is the backjump target. The backjump is then performed as a sequence of backward steps without making new decisions in the intermediate levels.

6.4 Type-uninterpreted search

This section presents an acceleration technique that does not fall into the conventional categories of CSP solving techniques. The *colouring technique* improves the representation of the CSP, because it eliminates a significant part of the variables. The idea is highly specific to ATPG-derived CSPs, and can be best performed in high-level descriptions.

Since an ATPG-based CSP is basically a fault simulation, the variables derived from circuit signals and the constraints derived from circuit components are present in two instances (Section 5.1.3), one belonging to the fault-free and the other belonging to the faulty domain. Considering a solution, we can easily realise that only those pairs may take different values that are affected by the injected faults.

If we knew in advance which are the signals that can be affected by the injected fault, we could inject an equality constraint between the two instances in order to make *implication* more powerful, or even better, not duplicate the variable at all. In addition, in multiple-output circuits there can be some signals that have no impact on the output where the discrepancy appears in the solution. If we knew in advance which these signals are, the generation of even the first variable instance could be avoided.

6.4.1 Node classification

Gate-level ATPG approaches (Section 3.2) include this idea. In some algorithms like D and PODEM, the nature of the algorithm guarantees that the discrepant D and $\bar{D}$ values cannot appear on signals before the fault location. Other approaches (Sziray's *composite justification* and Tilly's CONTEST) perform a dedicated preprocessing algorithm that reveals which signals have impact on the output and which may carry D or $\bar{D}$ values. Although at the gate level duplicated variables are handled as a single variable of the $\{0,1,D,\bar{D}\}$ domain, permitting Ds and $\bar{D}$s expresses the same concept.

Sziray's *node classification* algorithm (interpreted in the constraint environment) proceeds as follows:

1. The signals in the *coverage cone* of the selected output are classified as *relevant*, because only they can have an impact on the output. They will be generated in one or two instances in the CSP. The rest is labelled as *don't care*, and no copies of them are generated.

2. In the set of *relevant* signals, those that depend on the value of the faulty signal are classified as *potentially active*, and are generated in two copies in the CSP, since the

signal may take different values in the different domains during fault simulation. The remainder of the set is classified as *inactive*, and the corresponding variables are generated in a single instance in the CSP, since their values will not differ during fault simulation.

In programming terms, this algorithm can be implemented very easily by means of a backward token propagation beginning at the output, followed by a forward propagation beginning at the fault location (Figure 6.1).

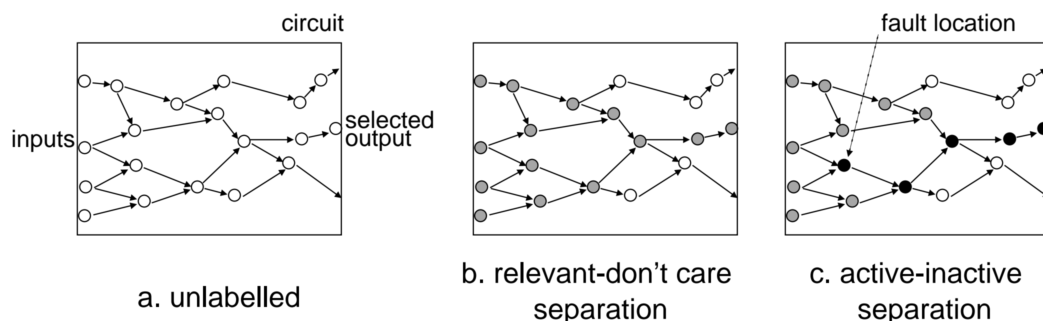

Figure 6.1: Phases of Sziray's node classification

The method is indeed a static preprocessing of the problem as long as there is only one output. For multiple-output circuits, we select one of the outputs where we expect the fault effect in the form of a discrepancy, which is a *decision* that can be wrong as well. The technique is applicable for sequential circuits too, but forward propagation must be performed from every time frame instance of the faulty signal. Thus the portion of variables to be duplicated will be much higher, resulting in a relatively small acceleration (Figure 6.2).

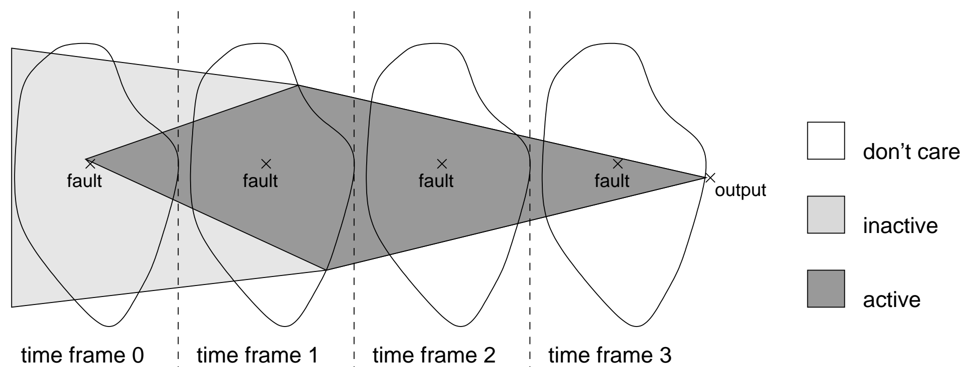

Figure 6.2: Large proportion of duplicated variables

6.4.2 Colouring goals

The idea of *type-uninterpreted search* or *colouring* has the same goals as Sziray's algorithm. However, it creates smaller CSPs for the same ATPG problem efficiently, because

it exploits the structural features of the circuit as well as component characteristics. The information required for this is available in compact form only in behavioural or hierarchical descriptions, but not in long gate-level netlist that are used by traditional ATPG algorithms.

The type-uninterpreted search phase exploits the fact that in high-level descriptions there are relatively few signals with large domains. Instead of assigning values from the domains of the signals to CSP variables, it assigns *tokens (colours)* from a small set to the signals in order to reveal which signals are relevant and which signals will carry different values during the fault simulation. Once a solution is found in the *colouring domain*, it has a chance to be a *typed solution*, a solution of the original CSP (Figure 6.3).

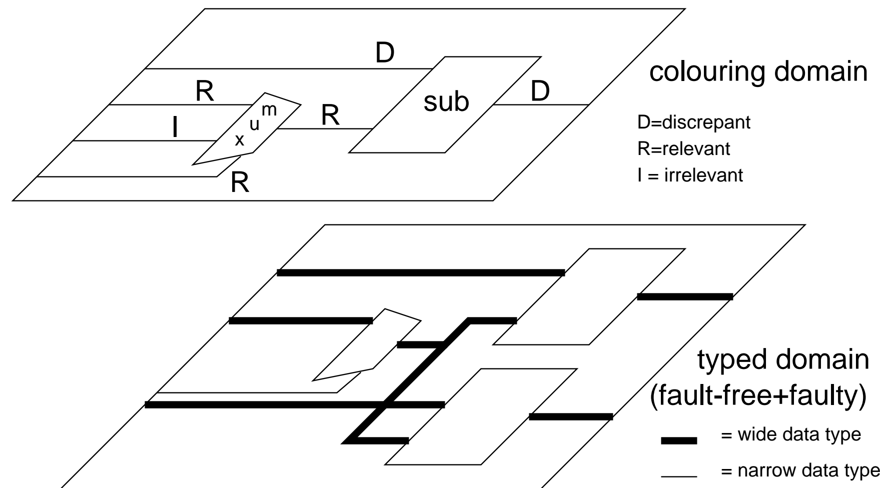

Figure 6.3: The colouring and the typed domains

Assigning tokens instead of typed values also appears in Csertán's approach [42] for early testability analysis. He uses tokens for the exploration of non-deterministic fault effects.

The type-uninterpreted phase of the BudaTest tool is no more a static preprocessing technique but the first part of the search. If a suitable solution is found in the type-uninterpreted domain, a second CSP with the identified number of variables and with the full variable domains is generated, and a typed search follows for the concrete test patterns. Since the second phase can fail, the high-level part of the search must be able to backtrack and generate systematically type-uninterpreted solutions. In other words, the two-phase algorithm proceeds as a single backtracking, but in some internal nodes of the decision tree (in the solutions of the high-level search), the entire problem representation is changed and a different CSP is generated (Figure 6.4).

6.4.3 Token semantics

The colour set appearing in the BudaTest algorithm is rather large. The variables of the type-uninterpreted domain can hold only a few of them. The semantics of these

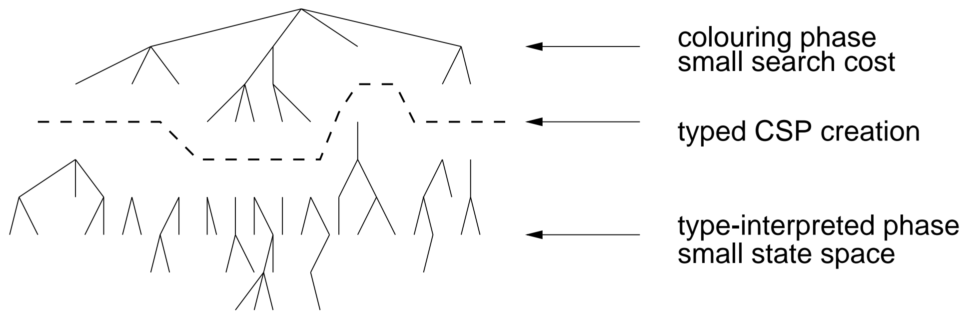

Figure 6.4: Two phases of CSP solving

colours directly correspond to the token set of the node classification algorithm, as they describe the relevance of the signal for the given fault.

- UNASSIGNED. In the course of the colouring process all unassigned variables are assigned a colour from the following set.
- UNKNOWN. This colour means that the value of the signal is irrelevant, therefore the typed CSP, generated at the boundary of the untyped and typed phases, will not even contain this variable.
- IDENTICAL. Due to the location of the variable or to the nature of the first decisions, this variable will have the same value in the fault-free and faulty typed domains, so it will suffice to generate it in a single instance, representing both domains.
- DISCREPANT. The variable may take different values in the typed domains, therefore it is generated in two instances.

Although the type-uninterpreted phase generally does not aim at assigning typed values to variables, in some distinguished cases (described at the discussion of colouring constraints), some additional information is recorded with the token. Examples for this include the following:

- a *typed value* if the token is IDENTICAL. This value will be set in the beginning of the typed search. Such values are not present for general data signals, only for those which are considered as control signals in some sense.
- a *Boolean flag* indicating whether a DISCREPANT pair *will* surely carry or only *may* carry different values (under the condition the CSP is solvable, i.e. a test exists). This latter flag is useful for subsequent heuristics, since settling down values for surely discrepant pairs should have smaller priority.
- an *index mask* indicating the bit positions where a DISCREPANT array signal is surely identical. The number of fault propagation paths can be reduced using this index when only a part of an array variable is used by a constraint.

6.4.4 Constraints in the colouring domain

The constraints derived from the components define a subset of the possible token combinations, like during the typed search. Due to the small type-uninterpreted domains table-based representation of the constraints is possible, which makes the *colouring implication* functions of the constraints rather simple and similar. In fact, the BudaTest tool uses the same colour checking function for every kind of constraint, although this function is highly customisable for the individual constraint. In addition, thanks to the object-oriented implementation of the entire system, the check function can be *overloaded*, so special components may have their individual accelerated consistency check and colour propagation function.

By default, each constraint has a table that describes its data propagation behaviour. These tables may contain more colours than listed in Section 6.4.3. The new tokens are not assigned to variables, but have an impact on what tokens are assigned. (This means that the *colouring intersection* operator is not commutative at all.)

A 2-to-1 multiplexer operating on any data type, for example, uses the following table:

output	selector	input 1	input 2
UNKNOWN	UNKNOWN (input)	UNKNOWN (input)	UNKNOWN (input)
EQUAL (to var 3)	IDENTICAL (typed 0)	EQUAL (to var 1)	UNKNOWN (input)
EQUAL (to var 4)	IDENTICAL (typed 1)	UNKNOWN (input)	EQUAL (to var 1)
DISCREPANT	DISCREPANT	KNOWN	KNOWN

Table 6.2: 2-to-1 multiplexer colour set

Every row contains 4 colours, one for each variable of the constraint. The first row is not specific to the multiplexer behaviour, because it is present in every component-derived constraint. It describes the case when the constraint is totally irrelevant, which can happen when the constraint has no impact on an output, or when its output is blocked and not used at all. The difference between UNKNOWN and UNKNOWN (input) is explained later (Section 6.4.7).

The remaining rows specify the data and discrepancy propagation features of the multiplexer, which primarily depends on the value of the *selector* input. Rows 2 and 3 describe that the condition of setting the value of the selector to 0 or 1, respectively, involves the equivalence of the tokens of the output variable and the selected variable whatever they be, including that of the additional information. In addition, the other variable can take any token, its propagation is blocked in the direction of the multiplexer. Hence, through multiplexers it is easy to establish a chain of unchanged values to propagate, or to block undesired propagation. To include EQUAL entries in colour tables wherever possible is very important because they allow the propagation of additional information, which means further significant reduction in the resulting typed CSP. If

we fail to propagate additional information due to the data transformation behaviour of the constraints, this auxiliary field is cleared.

The last row covers the case when we do not manage to force identical values on the selector in the two typed domains. KNOWN values mean that they allow IDENTICAL or DISCREPANT values on the values, but cause contradiction if they are already UNKNOWN.

There is a pre-search phase when the constraints can configure the behaviour of the variable assignment engine by setting certain static variable fields. This constraint, for instance, will explicitly prohibit the assignment of the IDENTICAL (without typed) value to its *selector* variable, because it has no corresponding row for it. This way it recognises that this variable is a distinguished control variable, the typed value of which is interesting even in the type-uninterpreted phase.

Another informative example for the constraint colour tables is that of a multiplier component that operates on wide integers:

product	factor 1	factor 2
UNKNOWN	UNKNOWN (input)	UNKNOWN (input)
IDENTICAL (typed 0)	IDENTICAL (typed 0)	UNKNOWN (input)
IDENTICAL (typed 0)	UNKNOWN (input)	IDENTICAL (typed 0)
EQUAL (to var 3)	IDENTICAL (typed 1)	EQUAL (to var 1)
EQUAL (to var 2)	EQUAL (to var 1)	IDENTICAL (typed 1)
IDENTICAL	IDENTICAL	IDENTICAL
DISCREPANT	DISCREPANT	KNOWN
DISCREPANT	KNOWN	DISCREPANT

Table 6.3: Multiplier colour set

Here row 1 is the general irrelevant case. Rows 2-5 represent "interesting" distinguished cases. Rows 2 and 3 describe the propagation blocking case by setting one of the input variables to 0. Rows 4 and 5, on the other hand, specify how to propagate tokens and assure they do not change. The remaining rows are general again and do not exploit special features of the multiplier, but are necessary for covering all cases. In the pre-search CSP solver configuration phase this constraint tells the solver that it may be worth assigning typed 0 and 1 IDENTICAL tokens to the corresponding variables, but does not enforce this. This demonstrates how control features of even the most generic data variables are recognised.

6.4.5 Correctness of the colouring search

After the introduction of the basic features of the type-uninterpreted search, we show that employing this search phase affects only the efficiency but not the results of the ATPG algorithm:

Theorem 3 *The CSP solver equipped with the colouring phase finds the same solutions as the one without colouring.*

Proof: We define the concept of *generalised restrictions* in a backtracking algorithm. A generalised restriction is some additional confinement in the composite domain of some variables, expressed in any form. The inclusion of *dynamic constraints* which are valid only below a certain decision tree level is also a generalised restriction.[2] *Dynamic constraint replacement* under a certain DT level is a generalised restriction too, as long as $C_{new} \subseteq C_{old}$.

Consider a "conventional" CSP solver which starts with the duplicated network shown in Section 5.1.3. We show that a colouring-equipped solution procedure is equivalent to backtracking in the conventional network.

As already discussed, backtracking is conditional search. A *decision* means that a solution is searched under the assumed condition. There is no risk of solution loss if the union of local decision candidates in a given DT node give back the unrestricted backtracking state the node represents.

The assignment of DISCREPANT colours represents no restrictions. IDENTICAL assignments mean the insertion of a dynamic equivalence constraint in the conventional network between the fault-free and faulty-domain variables. UNKNOWN assignments involve a constraint replacement in the conventional network with a more strict one: those row groups are deleted from the relation where the value of the UNKNOWN variable is relevant. Additional information assignments can be also expressed by restrictions in the conventional solution.

The union of the conditions the assigned tokens represent is the lack of restriction. Colouring implication effectuates only implied restrictions but does not create new ones, thus colouring is a normal backtracking in the conventional network. □

6.4.6 A colouring example

The gain of this token assignment phase can be well illustrated by showing how large the typed CSP can be with and without the untyped phase. Assume that we generate test for the GCD circuit (Appendix A), and that we expect a test length of 4 time frames. The "dumb" CSP, generated the way as described in Section 5.1.3 with the application of the iterative model (Section 3.2.4), will contain two copies of every variable and constraint shown in Figure 6.5, one participating in the fault-free, the other in the faulty circuit simulation. *Node classification* brings improvement, but still too many variables are *active*, because the first fault location is too close to the inputs (Figure 6.6).[3] In addition, all multiplexer inputs are considered relevant, even if it is always sure that relevant data come from the inputs in the first frame, and the effect of the fault in the first time frame

[2] In fact, the constraint literature often uses the concept of *dynamic constraints*. For example, the *learnt* information recorded by a solver (see *learning* in Section 4.3.3) is expressed by dynamic constraints.

[3] *Inactive* variables are shown with dashed line, while *don't care* ones are not shown at all.

is blocked. The next figure (Figure 6.7) shows how small the resulting typed CSP can be at the boundary of the two phases if we use colouring. Since the initial decisions of this phase select certain propagation paths and regard the others as irrelevant, the majority of variables are eliminated, and there are many variables present in one instance even if they could (but don't) depend on the value of a faulty signal. However, there are possibly many typed CSPs similar to that shown in Figure 6.7, because possibly wrong decisions precede the creation of the typed CSP. The decisions that lead to this typed CSP are also given in the picture.

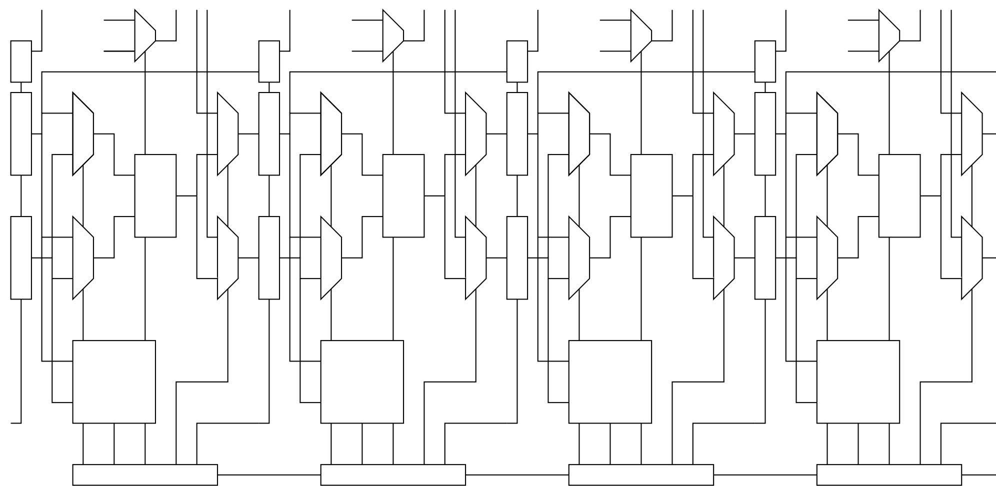

Figure 6.5: GCD in 4 frames, all variables in two instances

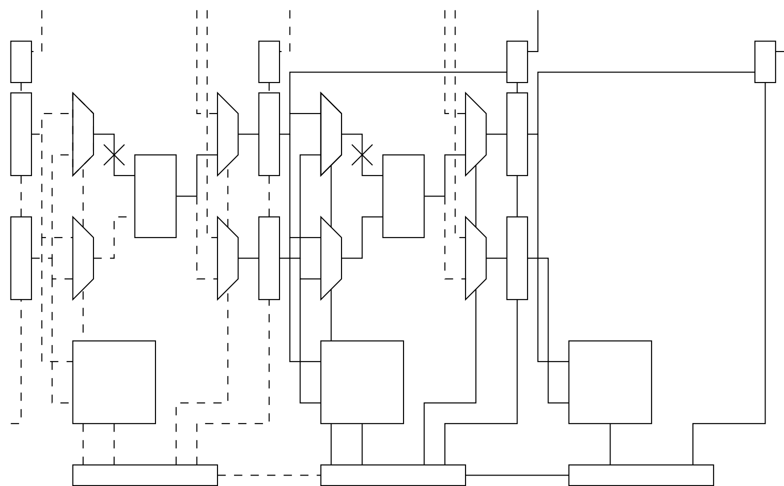

Figure 6.6: GCD in 4 frames, some variables eliminated by node classification

6.4.7 The handling of fan-outs, indexing and slicing

The difference between UNKNOWN and UNKNOWN (input) is whether they are *forcing* or not. An UNKNOWN table entry is always associated with component outputs, and

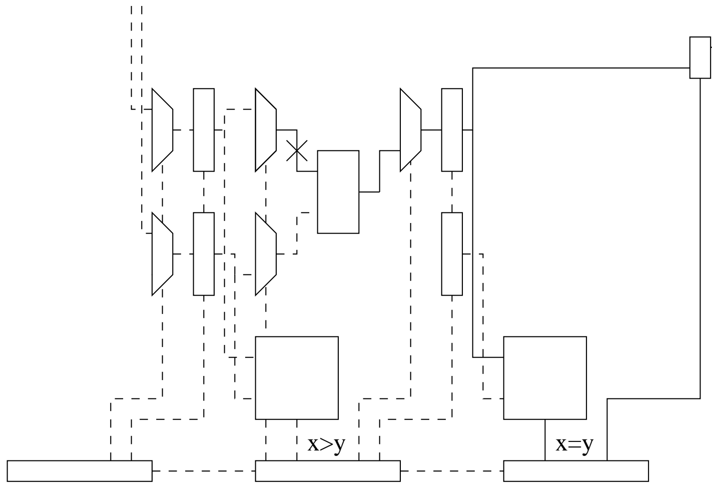

Figure 6.7: GCD in 4 frames, many variables eliminated by colouring

assigns an UNKNOWN token to the variable (or initiates backtrack if it is set otherwise). An UNKNOWN (input) can be consistent with other colours if the variable value may still be relevant, i.e. when it is a fan-out variable and it is input to several constraints. The effect of UNKNOWN (input) is illustrated in Figure 6.8. ID and DIS denote already assigned IDENTICAL and DISCREPANT colours. Assume that the colouring engine has just assigned an IDENTICAL (typed 1) colour to the variable *selector*. This will call the *implication* function of the multiplexer, which, in turn, will assign DISCREPANT to the multiplexer output and UNKNOWN (input) to the non-selected input. The latter assignment results in inconsistency in case *a*, because the BudaTest program avoids by force the unnecessary creation of irrelevant variables. In case *b* the value of the fan-out variable may still be required to satisfy other constraints, so the actual colouring can be part of a type-uninterpreted solution.

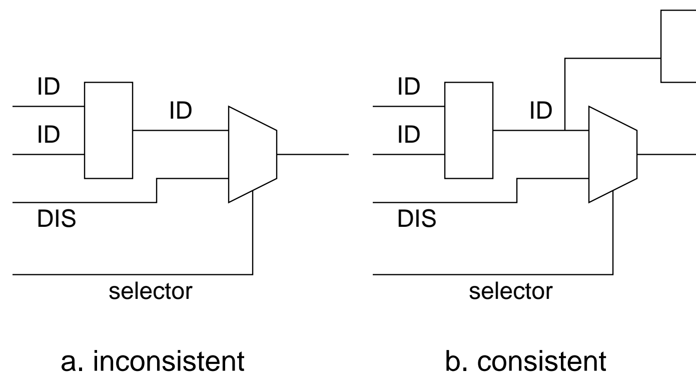

Figure 6.8: The effect of UNKNOWN (input)

Section 5.5 discusses the modelling technique of partial array referencing while Section 6.2.2 describes how they are related to the *mask* logic during the typed solution. The colouring phase also uses masks that specify in which position an abstract integer

or vector carries discrepant values. This mask is important when a discrepant variable is referenced, because an element or a part of an array is not necessarily discrepant.

For example, in Figure 6.9/a it is known that only the stuck-at position of the array carries discrepant values in the two simulations. This information is processed by the constraints that represent indexes and slices, so the DISCREPANT region will not grow unnecessarily (Figure 6.9/b). It is always preferable to propagate auxiliary information through IDENTICAL mechanisms so that the engine always know where the variables are exactly discrepant.

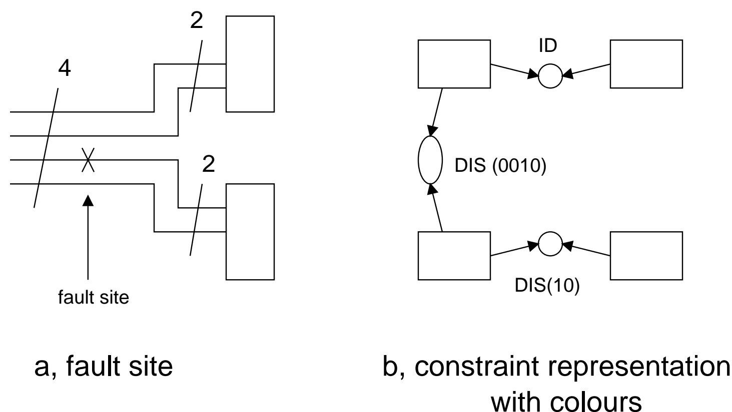

Figure 6.9: Auxiliary information used in D-propagation

6.5 Result evaluation

Since it is impossible to use analytical models of typical circuits, the efficiency of incorporated mechanisms will be examined by means of benchmarking, a common comparison technique in the ATPG area. The used benchmark set is described in Appendix A. In spite that the BudaTest approach is primarily targeted at architectural descriptions, the set includes the ISCAS'85 gate-level benchmark circuits [11] so that the evaluation of the "general" capabilities of the tool be possible. In accordance with the goals of this work, the emphasis will be placed on the architectural circuits of the set.

Unfortunately, no widely used high-level ATPG-purpose benchmark set exists, but FUTEG [2] participants have defined such a set. The set includes high-level synthesis (Section 2.2.2) benchmarks and high-level equivalents of some ATPG benchmarks, and represents various aspects and cases of digital circuit design: combinational or sequential, data- or control-dominated, simple or highly reconvergent etc.

The measurement report tables contain the following columns:

- *ratio of detectable faults* (when known): Some of the benchmark circuits, especially sequential ones, have a large proportion of undetectable faults. Small ratios are interesting because the time consumption shows how fast the algorithm traverses

the entire decision tree (Section 4.2.2). Moreover, when no CSP solution exists, forward heuristics have no impact on the time demand.

- *time-out:* To obtain reasonable response times, a time-out value is used frequently. The employed *time-out value* limits the number of decision tree nodes to $value*2^{16}$, then abandons the search.
- *time consumption:* The time demand of the CSP engine (including overheads like VHDL parsing, CSP creation etc) in seconds. The experiments have been performed on a 333 MHz PC-compatible.
- *total coverage:* The ratio of faults for which BudaTest has found a test. With no timeout it must equal the detectable coverage, otherwise it can be lower due to the possibly abandoned test generation phases.

No heuristics have been applied during the measurements, because the use of heuristic variable or value ordering could interfere with the evaluation of the pure effect of the presented techniques, which is the main purpose of this work. In every forward step, a variable adjacent to an already assigned variable was randomly selected. The order of assigned values is fixed and ascending. Because of the randomness of the results, average execution figures is presented.

In spite that BudaTest gave very good results for architectural benchmarks and fairly good results comparable to other tools for gate-level benchmarks, the lack of heuristic decision control also means that there are significant reserves, yet unexploited, in BudaTest.

6.5.1 Evaluation of implication-related techniques

Table 6.4 contains performance figures measured by the BudaTest tool with and without using implication and the presented mask and interval logic. The experiments show that implication is an absolute necessity, since the ATPG with implication switched off is acceptable only for the very low-end of the benchmark set.

Regarding the effect of the utilised data structures, we examine first the gate-level benchmark results (full adder, ISCAS set). As there is no opportunity to assign intervals and masks at the gate level, it is not surprising that the performance of the different representations were about the same. In consequence, the results are similar to CONTEST's unaccelerated results (Section 4.4), though a little worse due to the overhead of the high-level functions.

There are, however, differences in the benchmark set characterised by wide data and high-level data manipulation. The 4-bit adder is an interesting gate-level example where only the constraints responsible for the distribution of the input vector and for the collection of the output vector propagate masks, but this already brings significant improvement. For arithmetics-dominated circuits (gcd, bubble sort) the gain of interval

circuit	detect. ratio	time out	no implication		only bound		mask and interval	
			time	coverage	time	coverage	time	coverage
full adder	100.00%		12.33	100.00%	0.05	100.00%	0.05	100.00%
c432		5			2930	88.77%	3011	86.43%
c499		3			16646	50.08%	15889	51.88%
c880		3			210	99.31%	214	99.25%
c2670		1			62012	55.43%	59902	56.11%
4-bit adder (flat)	100.00%	1			17.77	100.00%	3.21	100.00%
4-bit adder	100.00%	1	11.73	75.48%	28.09	100.00%	8.22	100.00%
2-bit gcd (3 fr)	64.06%	5	111.5	3.21%	0.3	64.06%	0.36	64.06%
2-bit gcd (4 fr)	77.20%				20.57	77.2%	14.66	77.20%
4-bit gcd (3 fr)	66.34%				3.79	66.34%	3.04	66.34%
4-bit gcd (4 fr)	97.11%	5			247.8	76.93%	237.4	79.20%
8-bit gcd (3 fr)	67.93%	10			2110	40.21%	1577	61.35%
2-bit bubble (3 fr)	52.94%				2.98	52.94%	2.21	52.94%
2-bit bubble (4 fr)	77.20%	1			46.49	61.02%	37.83	68.38%
4-bit bubble (3 fr)	54.38%	5			1147	45.35%	695.5	54.38%
8-bit bubble (3 fr)	55.33%	3			17122	17.19%	9940	42.14%
2-bit multiplier	96.15%		8.87	96.15%	0.34	96.15%	0.08	96.15%
3-bit multiplier	94.82%	3	52.62	5.17%	43.75	32.75%	30.78	94.82%
4-bit multiplier	94.11%	10			711	46.02%	442	73.71%

Table 6.4: Implication performance for different techniques

logic is about 20-30% in time and 5-35% in coverage, and the difference in the coverage increases as the word length grows. In the circuits where partial array access is used, the acceleration due to the mask representation is also significant, and the increased fault coverage of the proposed representation is even more important.[4]

6.5.2 Backjumping performance

The performance results for backjumping are reported in Table 6.5. The results suggest that backjumping efficiency is related to the structural complexity of the circuits. In gate-level circuits the gain due to backjumping is indeed significant, although our results are still somewhat worse than those measured by CONTEST (Section 4.4). At the architectural level, however, components tend to be much fewer, and backjumping does

[4] An interesting experiment could be the use of various kinds of set logic not only in implication but in decisions as well. As the example of the *brute force* CSP solver algorithm shows, an abruptly swelling decision tree may entail too many dead-ends, because inconsistencies cannot be detected in internal DT nodes. When applied for variables of large domains, *set* (or *interval* or *mask* decisions could prevent the CSP solver from making too detailed decisions. Such a decision would not necessarily assign a specific value to a variable, but would merely restrict its domain. Of course, the strategy of these decisions could make up the subject of an entire dissertation.

circuit	detect. ratio	time out	without backjumping time	coverage	with backjumping time	coverage
full adder	100.00%		0.05	100.00%	0.05	100.00%
c432		5	2930	88.77%	1028	96.34%
c499		3	16646	50.08%	290	97.20%
c880		3	210	99.31%	14	100.00%
c1355		1	40711	21.32%	3558	95.80%
c1908		1	52932	47.44%	3100	90.20%
c2670		1	62012	55.43%	5310	88.20%
4-bit adder	100.00%		8.22	100.00%	7.14	100.00%
4-bit gcd (3 fr)	66.34%		3.04	66.34%	5.12	66.34%
4-bit gcd (4 fr)	97.11%	5	237	79.20%	221	82.10%
8-bit gcd (3 fr)	67.93%	10	1577	61.35%	2045	55.70%
4-bit bubble (3 fr)	54.38%	5	695	54.38%	88.3	54.38%
8-bit bubble (3 fr)	55.33%	3	9940	42.14%	6622	45.10%
3-bit multiplier	94.82%		30.78	94.82%	26.71	94.82%

Table 6.5: Backjumping performance

not bring orders of magnitude of improvement. There are even examples when the use of backjumping slows down the search.

We note that architectural backjumping seems to be promising when used with cutset-like heuristics (Section 4.3.2, [53]), because they create many independent decision sites in the CSP.

6.5.3 Evaluation of the type-uninterpreted search technique

We claim that many small searches can be better than one big search. Why do we expect better results with colouring?

Assume for simplicity's sake that the CSP without colouring has N variables and each variable has a uniform domain size d. The state space consists therefore of d^N elements. What happens if we increase the domain size? A new domain size of kd will produce a state space of $k^N d^N$ elements.

Now let us make the same calculation with colouring. Assume the we find c colouring solutions, and assume they all exclude N_E and create the typed CSP with $n = N - N_E$ variables. Thus, the state space of the combined phases will be cd^n elements large, which grows less dramatically if we increase d. This shows that colouring is preferable for high-level circuits containing wide data. In addition, this reasoning applies to the full traversal of the DT (e.g. when no test exists) but not for a best-first goal node oriented search. Since a colouring solution is more likely to lead to a typed solution than other

typed decisions, we expect that it finds a test pattern sooner.

Table 6.6 provides comparative results on the time demand and fault coverage of the CSP solver with node classification and colouring.[5]

circuit	detect. ratio	time out	node classification		colouring	
			time	coverage	time	coverage
full adder	100.00%		0.05	100.00%	0.21	100.00%
4-bit adder (flat)	100.00%	1	3.21	100.00%	452.5	77.19%
c432		5	2930	88.77%	12321	23.64%
4-bit adder	100%		8.22	100.00%	3.69	100.00%
8-bit adder	100%	5	213	87.87%	264	89.39%
3-bit multiplier	94.82%		30.78	94.82%	3.98	94.82%
4-bit multiplier	94.11%	5	276	51.96%	94.9	92.15%
2-bit gcd (5 fr)	93.75%	5	56.71	87.50%	72.34	93.75%
4-bit gcd (5 fr)	97.11%	5	560.8	73.07%	15.86	97.11%
8-bit gcd (5 fr)		5	2991	60.11%	632	93.12%
2-bit bubble (3 fr)	52.94%		2.98	52.94%	67.61	52.94%
4-bit bubble (3 fr)	54.38%	5	695.5	54.38%	126.1	54.38%
8-bit bubble (3 fr)	67.93%	10	3430	43.10%	811	63.49%

Table 6.6: Effect of node classification and colouring

The results can be interpreted as follows. Related to node classification, the performance of the two-phase search with colouring is significantly worse for the gate-level benchmarks, and is significantly better for the high-level benchmark circuits. The acceleration due to colouring correlates with the degree of abstraction.

This is exactly what we expected. Colouring slows down CSP solving in gate-level benchmarks because variable domains do not become smaller (they even become larger), there are no structured propagation paths to explore, and that repetitive CSP duplication/disposal involves much overhead.

The situation changes rapidly as we advance upward in the abstraction level, that is, as control and data signals become separated, clusters of gates are specified as a single component, and signals become increasingly wider. The examples of the *bubble sort* and *greatest common divisor* circuits clearly show that the performance of the two methods is comparable for small word lengths, but colouring brings much more improvement when we increase the data width. This is due to the state space size of the type-uninterpreted phase which remains constant even if the data width grows.

Because of the current trends in the scale and methodologies of circuit design (see

[5]Backjumping support was removed from the BudaTest version which was used to measure the colouring efficiency, because it did not provide sufficiently good results for architectural circuits, and because the continuous maintenance of variable lists made the program very error-prone.

Section 2), I am convinced that the second group of the benchmark set represents today's practical ATPG environment. Although the NP-completeness and the overall exponentiality of the ATPG problem is unquestionable, the size of circuits tackled successfully by intelligent problem solvers can be still significantly larger than one could expect considering simply the number of gates in a design.

Chapter 7

Manufacturing test on the wafer

Besides ATPG, the other time demanding procedure is the execution of precalculated tests after manufacture in the VLSI industry. The current automatic test equipment (ATE) based technology follows the procedure below:

1. If there are no more untested chips on the wafer, READY.
2. Position on the next untested chip.
3. Drive the chip input pins with the test patterns and measure the outputs produced by the chip. Compare the results with the reference results, and classify the chip as fault-free or faulty accordingly.
4. Go to Step 1.

Unlike the time needed for ATPG, the time consumption of this sequential test execution is proportional to the number of produced chips.

A recent idea coming from the field of self-diagnosis [63] is to regard the chips on the wafer as a single system, to compare the test results by comparators built in between the chips, and to evaluate the syndrome downloaded from the comparators. If the total number of faulty chips does not exceed an algorithm-dependent limit, then the diagnosis algorithm correctly identifies the state of the "components" of the system, i.e. the chips themselves. This method would transform sequential testing into parallel and would reduce the time required for test execution into its fragment. In addition, the costly ATE could be avoided.

A major obstacle in the application of this method is its high sensitivity to the manufacturing faults possibly occurring in the additional circuitry. A faulty comparator could invalidate not only the classification of the immediately adjacent chips, but in unlucky cases that of all the chips.

Our goal in this work is to implement a dedicated test technique that eliminates the risk of test invalidation caused by the faulty diagnosis-purpose circuitry. We will show that the technique tolerates multiple stuck-at faults of the comparator and of the syndrome collection circuitry.

7.1 Diagnosis terms and wafer-scale testing

System-level diagnosis, also called self-diagnosis, has been introduced by Preparata et al. in 1967 [64]. In self-diagnosis a system, composed of several units connected by bi-directional links, can be diagnosed using *tests* performed by the units themselves. Each test involves two units, called the *testing* and the *tested* units, and proceeds as follows:

- the testing unit requests the tested unit to run a test;
- the tested unit returns a result to the testing unit;
- the testing unit compares the actual and the expected results, and generates a binary test outcome. The outcome is 0 if the actual and the expected results match (the test passes), 1 otherwise (the test fails).

The collection of the test results is called the *syndrome.*

The test results are not necessarily reliable, since testing units themselves may be faulty. Different hypotheses upon the test outcome generated by faulty units lead to different *invalidation rules*, and consequently to different *diagnostic models.* The most widely used diagnostic model is the Preparata-Metze-Chien (PMC) model introduced in [64], which assumes arbitrary test outcomes for tests performed by faulty units. The invalidation rule of the PMC model is shown in Table 7.1.

testing unit	tested unit	test outcome
fault-free	fault-free	0
fault-free	faulty	1
faulty	fault-free	0 or 1
faulty	faulty	0 or 1

Table 7.1: Invalidation rule in the PMC model

A less general diagnostic invalidation rule is the BGM model [65]. When this model is valid, the faulty testing units cannot produce 0 outcomes.

Comparison models have also been introduced in the literature for systems composed of identical units. In comparison models test results are gained from built-in comparators which compare the outputs of adjacent units that run the same test. Again, different assumptions on the behaviour of the faulty units and comparators lead to different diagnostic models. In Malek's model [66] the comparators are implicitly assumed to be fault-free and the comparison outcome is 0 only if both the compared units are fault-free. This can be considered as the comparison-based equivalent of the BGM model. The model introduced by Chwa and Hakimi [67], which also assumes reliable comparators, allows arbitrary comparison outcomes when both the compared units are faulty, like the PMC model for bidirectional tests. In [68], Maeng and Malek proposed a modification

of Malek's model, in which comparisons are performed by the system units themselves: if unit i is adjacent to units j and k, then i may be utilised to compare the outputs of j and k. This requires the system units to be homogeneous and able to perform comparisons, which condition is satisfied when the units are processors.

Rangajaran, Fussel and Malek [63] suggested that system level diagnosis may find application in the testing of VLSI chips on the wafer during manufacture (*wafer-scale testing*). In this case the main goal of the diagnosis is the identification of good integrated circuits (ICs) within the wafer, which will be packaged, while the faulty circuits will be discarded. This approach reduces the time needed to test the chips as well as the cost of ATE-based testing (see Section 1.2.3).

If the nature of the manufactured chips is unrestricted, the comparison model is the best choice for this application. In order to implement self-diagnosis on the wafer, hardware support is necessary. First, interconnection links need to be introduced and comparators be wired on each link. To keep the number of comparators relatively small, the interconnection structure should be regular and the degree of nodes small. Furthermore, hardware support is required to provide the test sequence for the chip inputs and to collect the outcomes of the comparators. The latter task may be executed by the probing unit, which reads the outputs of the comparators and transmits them to a reliable external computer.

7.2 Evaluation of the syndrome

Once the external computer receives the syndrome, it executes the diagnosis algorithm. The output of this algorithm is the diagnosis of the wafer, i.e. the classification of each individual chip as fault-free or faulty. The correctness of the diagnosis depends on the validity of the applied comparison model.

The algorithm is *complete* if every component is identified as fault-free or faulty, and is *correct* if no faulty chip is diagnosed as fault-free and no fault-free chip is diagnosed as faulty. It is known that a correct and complete diagnosis is granted only when the number of faults in the system is fewer than the so-called *diagnosability*, which is bounded above by the minimum degree of the nodes in the diagnostic structure [69]. Since in practical applications like wafer-scale testing this is not true, some algorithms target only completeness [70, 71, 72, 73], but may contain errors with low probability. Maestrini et al. [74] aim at the proven correct diagnosis with a non-guaranteed completeness.

Although the fault-tolerant issues discussed here are not specific to any particular diagnosis algorithm, we cite Maestrini's algorithm (tailored to the Chwa-Hakimi comparison model) to give an example how the syndrome can be decoded. We call those adjacent chips 0 (1)-connected whose built-in comparator shows that the test output sequence equals (differs).

- First, surely faulty units are identified. This is performed by finding chains of 0-

connected chips where a 1-comparison exist between any two elements. The chips in a chain are faulty, since they are all in the same state due to the 0-connections, and this state cannot be fault-free due to the 1-connection (Figure 7.1). The remaining chips are classified as *zero* chips.

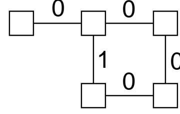

Figure 7.1: Faulty chips

- Then, disjoint pairs of 1-connected *zero* chips are searched and labelled as *dual units*. The goal of this step is to trade a faulty unit for an unknown one, as one of them is surely faulty. This way more reliable assumptions can be made about the number of faults in the remaining part.

- The remaining *zero* units are *aggregated* following 0-connections, and the largest aggregate is selected as *fault-free core* (FFC). If several largest aggregates exist, the FFC is the union of them.

- In the last step of the algorithm, the fault-free core is extended recursively, relying on the test results made by the units in the FFC.

Apart from the validity of the diagnostic model, the correctness of every diagnosis algorithms requires that the number of faulty components do not exceed a certain limit. In the case of Maestrini's algorithm, this limit is asserted by the algorithm itself based on the cardinality of the sets composed during the steps of the procedure. His general diagnostic algorithm, developed for toroidal grid structures and bidirectional (not comparison-based) tests, has the favourable property that the fault limit is bounded from below by the *syndrome-independent bound*, which is $O(N^{2/3})$, where N is the number of the chips on the wafer [75].

7.3 Impact of comparator faults

Like the presented algorithm, comparison-based algorithms highly depend on the validity of the following statements, which are consequences of the Chwa-Hakimi model [67]:

- Two 0-connected chips are in the same state.

- Among two 1-connected chips there is a faulty chip.

Consequently, it leads to invalid diagnosis if the comparators, containing physical faults, produce incorrect results. Table 7.2 describes how the Chwa-Hakimi model is violated in the presence of comparator faults.

case	chip 1 state	chip 2 state	comparator state	comparison result	violated
a	good	good	any	0	no
b	good	good	faulty	1	yes
c	good	faulty	any	1	no
d	good	faulty	faulty	0	yes
e	faulty	faulty	any	any	no

Table 7.2: Violations of the Chwa-Hakimi model

Note that *masked* comparator faults, i.e. when faulty comparators produce model-conform results, do not present a problem in the wafer-scale testing application, because the comparators will never be used again after the diagnosis is performed. On the other hand, *active* faults (cases *b* and *d*) may invalidate the diagnosis.

Case *b* is not critical in itself, because a correct diagnosis can still be obtained by means of a certain degradation of the diagnosis algorithm. The degradation consists in disregarding the second assumption about 1-connections, which prevents the algorithm from falsely diagnosing a cycle of good units and one faulty comparator (e.g. in Figure 7.1 if the 1-connection is caused by a faulty comparator). Since there is no way to prove that a unit is faulty without this assumption, the diagnosis result is a set of fault-free chips and a set of *suspicious* chips.

Case *d* poses a much more serious problem. The correctness of every diagnosis algorithm depends on the validity of the *aggregation* step, which is based on the assumption about 0-connections. The possible occurrence of case *d* could entirely invalidate the diagnosis, since an erroneous 0 comparison may lead to the aggregation of fault-free and faulty parts.

7.4 Pre-diagnosis comparator test session

In this section a preliminary test session is proposed in order to detect the situation described by case *d*. This session is different from the normal diagnostic test in the sense that adjacent units are fed by different input values. We assume that a faulty unit always produces the same response for the same input vector, even if the design is sequential, and that arbitrary patterns can be propagated to the output of a fault-free unit. These requirements can be fulfilled by a special wafer design shown in Section 7.7.

Our goal is to test the comparator between a fault-free and a possibly faulty unit exhaustively. If both units are good, this can be carried out without any difficulty. Faulty units, however, may fail to feed the comparator with the required test patterns, thus the fault of a unit may mask the fault of the comparator.

We examine a single-bit slice of the comparison first. Figure 7.2 illustrates the

behaviour of the adjacent units during this session when every component is fault-free (Fig. 7.2/a) and when one of the units, say B, and the comparator are perhaps faulty (Fig. 7.2/b). The possibly faulty B produces an $xyxy$ output sequence for the four test vectors.

Table 7.3 shows the effect a faulty B unit may have on the comparator test. If B does not alter the comparator test patterns (line 1), then the comparator will undergo an exhaustive test. If B produces the same responses for in_0 and in_1 (lines 2 and 3), then the combinational comparator will produce some $aabb$ sequence but surely not the expected 0110. Line 4 is, however, problematic, because the faulty unit B negates its responses for in_0 and in_1, and the faulty comparator might invert them again, thus passing the test in spite of faults, which could violate the diagnostic model assumed by the diagnosis algorithm. Fortunately, it can be easily seen that in most practical single-bit comparator designs, e.g. in the one shown in Figure 7.3, no combination of stuck-at faults produces such a behaviour.

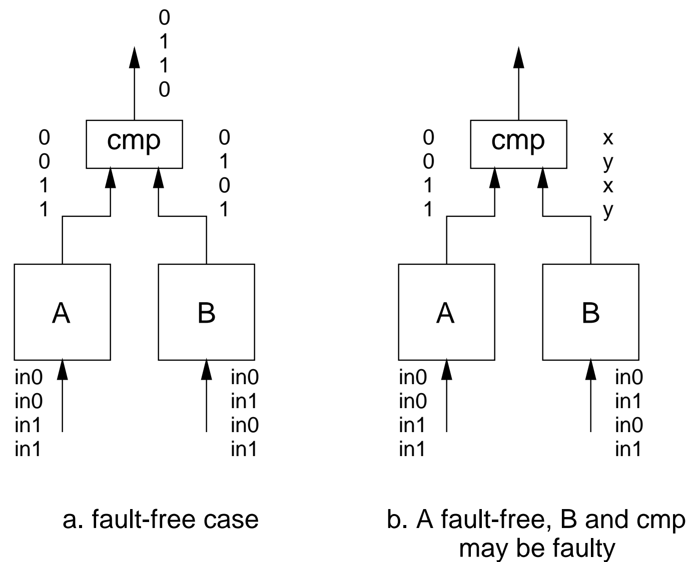

Figure 7.2: Comparator slice test session

x	y	effect
0	1	full comparator test
0	0	no 0110 output sequence
1	1	no 0110 output sequence
1	0	problematic

Table 7.3: Impact of the behaviour of B

Lemma 1 *In the 1-bit comparator in Figure 7.3, the inverting behaviour of the comparator cannot occur for any multiplicity of gate-level stuck-at faults.*

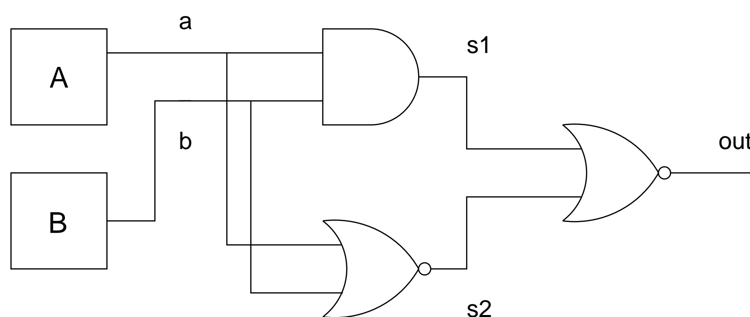

Figure 7.3: A simple 1-bit comparator

Proof: Table 7.4 lists all stuck-at fault combinations and shows their corresponding detection patterns.[1] Some fault combinations (deriving from *don't care* entries) require two input patterns to be detected. Although the faulty B may fail to drive the expected values to line b, this case will still produce discrepant outputs. Since in every row the value of b is constant, the erroneous behaviour of B is detected by detecting an equivalent additional stuck-at fault of line b. For example, in row 3 the pattern 11 detects the case of line s_1 stuck at 0 combined with any assignment of faults to *don't care* entries, except s_2 stuck at 1 or, equivalently, both a and b stuck at 0. The input pattern 01 is needed to handle such exceptions. The erroneous behaviour of B is equivalent in this row with the fault of b *s-a-0*, which is covered by assigning *s-a-0* to the *don't care* entry of column b.

out	s_1	s_2	a	b	pattern (ab)
0					01
1					00
ff	0				11 and 01
ff	1				01
ff	ff	0			00 and 10
ff	ff	1			01
ff	ff	ff	0		10 and 00
ff	ff	ff	1		01 and 11
ff	ff	ff	ff	0	01
ff	ff	ff	ff	1	00

Table 7.4: Single-bit comparator faults and detecting patterns

The slice test can be easily extended to an arbitrary comparator width while the test length is only a linear function of the number of slices.

[1] The table entries have the following meaning: 0 (1): s-a-0 (s-a-1) fault; ff: fault-free state; empty box: any state (don't care).

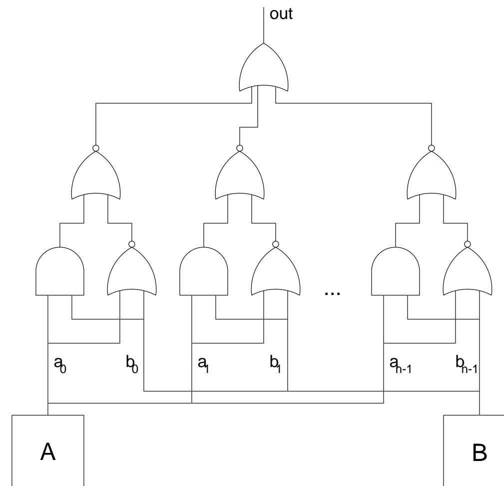

Figure 7.4: A simple n-bit comparator

Theorem 4 *Consider the n-bit comparator of Figure 7.4. Assume that A is fault-free and the possibly faulty B produces identical responses for identical inputs. Then, the test patterns in Table 7.5 will detect any multiplicity of stuck-at faults in the comparator combined with any faulty behaviour of unit B.*

a_0	b_0	...	a_i	b_i	...	a_{n-1}	b_{n-1}
0	0		0	0		0	0
0	1		0	0		0	0
1	0		0	0		0	0
1	1		0	0		0	0
...	...		...	...		...	...
0	0		0	1		0	0
0	0		1	0		0	0
0	0		1	1		0	0
...	...		...	...		...	...
0	0		0	0		0	1
0	0		0	0		1	0
0	0		0	0		1	1

Table 7.5: Test patterns for n slice

Proof: The table contains a 4-vector long test sequence for each comparator bit slice, line 0 (a shared vector for each slice), and lines $3i + 1$ to $3i + 3$ for the ith slice ($i = 0, ..., n-1$). They trivially detect any fault combination involving a stuck-at fault on line *out*. For the remaining faults, we will show that the test of slice i can be performed in spite of the effect of faults in other slices. Two cases may occur during the slice test:

- As an effect of a fault in B or in the comparator, a faulty 1 value appears on some OR-gate input line belonging to a slice other than i. Call the vector producing it v. The four vectors contain another one (w) where only a_i differs from v. The slices other than i will receive the same actual input values (even if modified by B) when we apply v and w, therefore the comparator output will be 1 for both patterns. On the other hand, the fault-free outputs should be different for v and w, i.e. the fault is detected.

- The OR-gate input lines belonging to slices other than i always hold the value 0. In this case, the presence of faults in unit B or in the other comparator slices does not disturb the test of slice i, which detects any stuck-at fault in the slice according to the previous lemma.

Repeating the test for all slices, every combination of stuck-at faults in the comparator will produce at least once an output different from the expected one. □

From the point of view of the diagnosis algorithm, we can summarise these results the following way: *A comparator passing the proposed test does not contain stuck-at faults, therefore the aggregation step can be done safely. Case* d *of Table 7.2 is eliminated.*

It is also true that the comparator test session excludes the undetected occurrence of case b, thus the algorithm can rely on the 1 results of the comparators that passed the test. However, few reliable 1 results will be generated without additional techniques, because a fault-free comparator is unlikely to pass the test if one driving chip is faulty and it fails to provide the expected test vectors for the comparator. On the other hand, if the comparator is fault-free and the fault in the chip does not modify the comparator test patterns, then the comparator test will pass, and the comparison result will be a reliable 1. The wafer implementation suggested in Section 7.7 highly exploits this feature.

7.5 Fault-tolerant result observation

The bottleneck in the performance of comparison-based testing is the observation of comparison results. Since the diagnostic test can be very long, the diagnoser should not be required to read the entire sequence of comparator outputs; instead, a syndrome collection circuitry should be wired to every comparator. If we disregard possible faults in the diagnostic circuitry, this could be a 1-bit RS flip-flop, which is reset before executing the diagnostic test, and set by any mismatch the compared chips produce.

This technique can also be used in the approach extended with a comparator test session, but some problems must be solved. First, a simple 1-triggered flip-flop is not sufficient, because the comparator produces 1 results during the comparator test session even if everything is fault-free. Furthermore, potential faults in the flip-flop must be taken into account, therefore the comparator test must be extended to cover faults in

the collection circuitry as well. Finally, the number of read-out operations should be minimised, because they are performed sequentially.

I will present a syndrome collection strategy and a general overview of the supporting hardware which handles all these problems. An RS flip-flop (Figure 7.5) can still be used for syndrome collection, with the following additional features:

- It contains a *guard* signal, controllable by the external diagnoser, which masks the transients that occur on the *set* input of the flip-flop.

- An auxiliary signal *aux*, also controllable, is used to control the *set* line. During the comparator and diagnostic test sessions, this line always holds the same value as the expected output of the comparator, so that one flip-flop is sufficient for collecting both 1s and 0s. Furthermore, during the flip-flop test (see later), the signal improves controllability.

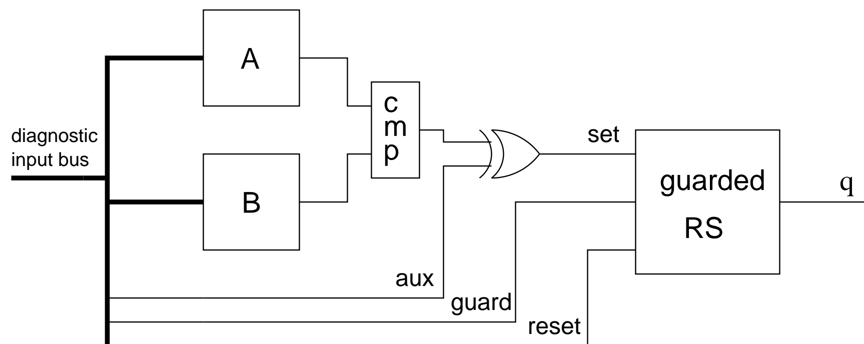

Figure 7.5: Syndrome collection circuitry

We extend the comparator test session to include a flip-flop test, which, if passes, ensures that the collection circuitry is also free from faults. We will prove that the flip-flop test detects *all* combinations of stuck-at faults in the collection circuitry as well.

The complete test session, including the extended comparator test session and the comparison-based diagnostic test session, requires only 3 read-out operations (Figure 7.6, and proceeds as follows.

- Initially, we apply the first vector of Table 7.5, we set *aux* to 1, and disable *guard*. This should make line *set* hold value 1 in the fault-free case. We issue a *reset* command, then enable *guard* (which should set the output, since *set* still holds 1), and check if the FF has been indeed set.

- Next, we disable the *guard*, issue a reset again, and perform the comparator test, taking care that *aux* always holds the same value as what is expected from the comparator, and that *guard* is only active when the comparator output settles down. After the comparator test, we give an impulse to *aux* without enabling the *guard*, and read out the comparator test result, which is expected to be 0.

- Finally, we perform the diagnostic test, keeping *aux* 0 and enabling *guard* after each vector, and read out its result.

The next theorem says that the result of the diagnostic test is reliable if we receive a 10 sequence for the first two read-outs.

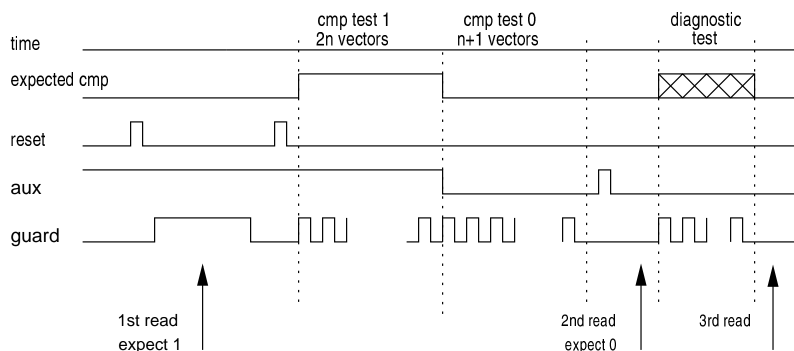

Figure 7.6: Complete test

Theorem 5 *For the gate level model shown in Figure 7.7, all combinations of gate-level stuck-at faults in the syndrome collection circuitry will cause the combined flip-flop–comparator test to fail.*

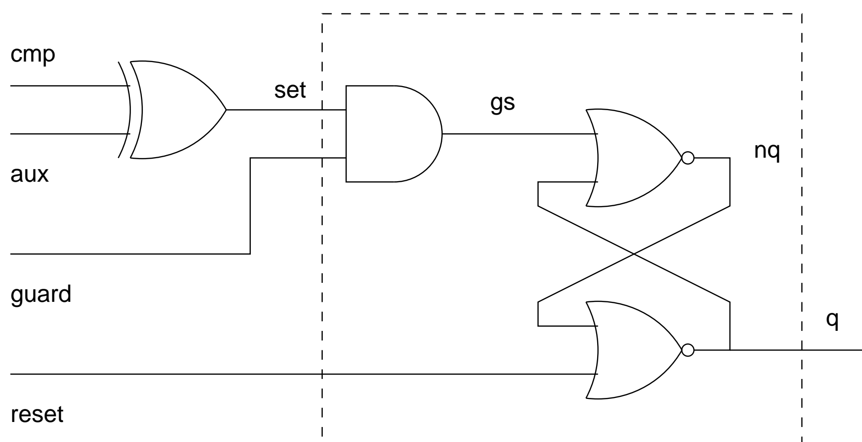

Figure 7.7: Syndrome collection at the gate level

Proof: We show in particular that every combination of stuck-at faults entails a test result other than 10. Table 7.6 lists all stuck-at fault combinations along with the possible read-out results for the first two read operations. The meaning of the table entries is the same as in Table 7.4.

Some faults may produce several output sequences, depending on the actual behaviour of the driving circuitry (A, B, and the comparator), and on the signals of which the state is listed as *don't care* in the given line. For example, the line where signals q, *reset*, *nq*, *gs*, and *set* are fault-free but *aux* is s-a-0 summarises the following:

q	reset	nq	gs	set	aux	guard	eq. fault	read out
1								11
0								00
ff	1						q s-a-0	
ff	0							0X or 11
ff	ff	1					q s-a-0	
ff	ff	0						00
ff	ff	ff	1				nq s-a-0	
ff	ff	ff	0					00
ff	ff	ff	ff	0			gs s-a-0	
ff	ff	ff	ff	1				00 or 11
ff	ff	ff	ff	ff	0			0X or 11
ff	ff	ff	ff	ff	1			0X or 11
ff	ff	ff	ff	ff	ff	0	gs s-a-0	
ff	ff	ff	ff	ff	ff	1		11

Table 7.6: Gate level fault coverage of the test

- If the *guard* is s-a-0, the read-out sequence will be 00.
- If the *guard* is s-a-1 or fault-free, and the comparator correctly produces 0 for the initial pattern, then the first bit read out will be 0.
- If the *guard* is s-a-1 or fault-free, and the comparator incorrectly produces 1 for the first vector of Table 7.5, then the read-out sequence will be 11, since the first vector pattern is part of the comparator test, which will therefore fail.

Note that we exploit again the fact that the circuitry comprising the comparator and the two units behave combinationally during the comparator test (see the implementation granting this feature in Section 7.7). Since no fault set listed in the table allows for a 10 read-out sequence, a passed FF–comparator test guarantees that the circuitry is free from stuck-at faults. □

7.6 Fault tolerant comparison model

The introduction of the comparator test phase allows for a refinement in the comparison model. The ternary outcome of the complete test is determined according to Table 7.7.

In practice, diagnosis algorithms are easily modified to handle *unreliable* outcomes. Maestrini's algorithm, for instance, works correctly if it does not declare chips faulty based on unreliable 1-connections and does not aggregate along unreliable 0-connections.

test outcome	interpretation
100	0-connection
101	1-connection
other	unreliable

Table 7.7: Refined comparison model

7.7 Wafer implementation

To carry out the proposed additional test sessions, the wafer diagnosis circuitry should meet the following requirements.

- The fault-free circuits should be able to provide the comparator with every comparator test pattern of Table 7.5.
- A circuit containing permanent faults should produce identical responses for identical inputs.
- Although the units are fed by the same input during the diagnostic test session, during the comparator test phase two adjacent units should be driven with different input values.

The first two conditions, if not already fulfilled by the actual unit design, can be easily satisfied by a simple modification in the circuit design, e.g. by multiplexing the output of the unit.

The third condition requires either the input bus in the wafer diagnosis circuitry to be duplicated, or the chips on the wafer to be non-identical. For instance, in a rectangular grid diagnosis structure there must be two kinds of chips.

A comprehensive solution to these problems could be the addition of some simple combinational logic that generates the comparator test patterns for the comparator test session (Figure 7.8). One bit, encoding the position of the chip ($A/\bar{B}$), is introduced only in the final stage of the design of the wafer masks. During the comparator test session the diagnosis circuitry receives a wafer-wide signal which orders it to use the output of the test pattern generator (TPG) instead of that of the unit.

The function of the TPG can be implemented with a small ROM module indexed by the common input, or with a more concise combinational logic, as the test patterns are easily compressible. It is important that a sequential design for the TPG logic would contradict the assumption that a repeated test pattern is always altered the same way by permanent faults.

This solution has a number of favourable features:

- The chips can play the role of either unit A or B without the intervention of the diagnosis algorithm and without requiring the duplication of the input bus. The

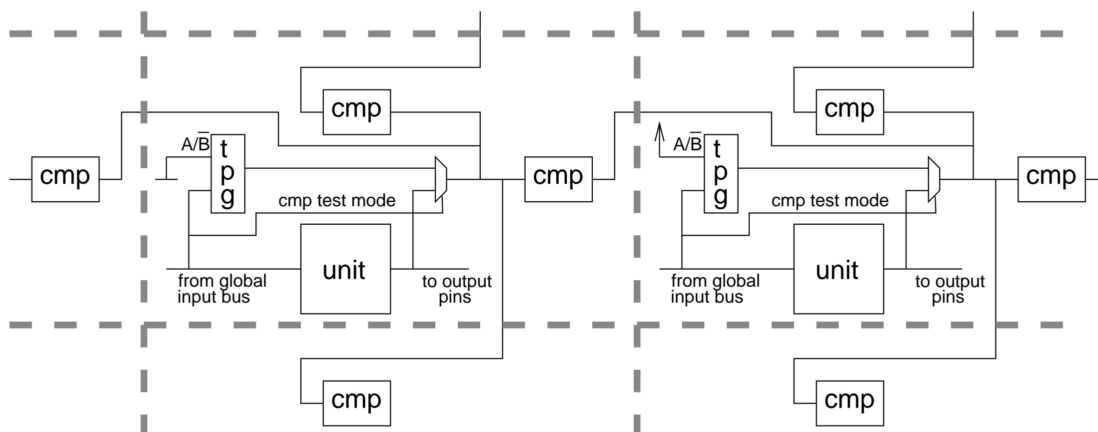

Figure 7.8: Non-identical chips on the wafer

additional cost (the extra surface) is small, especially in the case of complex chips.

- The wafer-based testing implementation remains hidden to the circuit designer, because the TPG and comparator subcircuitry can be added by the silicon factory, and the circuit user, because during packaging diagnostic links are not connected to pins.
- The design of the TPG and the comparator is independent of the nature of the IC to be manufactured, since including this additional logic does not require information other than the output width.

An even more desirable property of the TPG implementation is the satisfactory treatment of case *b* in Table 7.2. A fault-free comparator will pass the comparator test and will produce a reliable 1 outcome if one driving IC is faulty but the TPG logic is intact, which has relatively high probability.

Bibliography

[1] H. Fujiwara: Logic Testing and Design for Testability, MIT Press, Cambridge, Massachusetts, 1985

[2] FUTEG: Functional Test Generation and Diagnosis. Copernicus project CP94 - 9624.

[3] E.B. Eichelberger and T.W. Williams: A Logic Design Structure for LSI Testability, *Proc. of the 14th Design Automation Conf.,* New Orleans, 1977, pp 462-468.

[4] K.P. Parker: The Boundary-Scan Handbook: Analog and Digital, Kluwer, 1998

[5] P. Malinverni: Long-term Research Aspects of Circuit Design and Tests, Invited Keynote Speech, *IEEE European Test Workshop,* Cagliari, May 1997

[6] Cadence Design Systems. `http://www.cadence.com`

[7] Synopsys Design Tools. `http://www.synopsys.com`

[8] Mentor Graphics Products. `http://www.mentor.com`

[9] I. Park, K. O'Brien and A.A. Jerraya: AMICAL: Architectural Synthesis Based on VHDL, *Synthesis for Control Dominated Circuits (A-22),* ed. Saucier and Trilhe, Elsevier Science Publishers (North-Holland), 1993, pp 219-234

[10] IEEE Standard VHDL Language Reference Manual, IEEE Standard 1076, 1988

[11] F.H. Brglez and H. Fujiwara: A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran, *Proc. Int. Symp. on Circuits and Systems,* 1985, pp 663-698

[12] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho and G. Shurek: Test Program Generation for Functional Verification of PowerPC Processors in IBM, *Design Automation Conf.*, 1995

[13] S. Mourad and E.J. McCluskey: Fault Models, in: *F. Lombardi, M. Sami (ed): Testing and Diagnosis of VLSI and ULSI,* Kluwer, 1988, pp 49-69

[14] H.K. Lee and D.S. Ha: SOPRANO: An Efficient Automatic Test Pattern Generator for Stuck-Open Faults in CMOS Combinational Circuits, *ACM/IEEE Design Automation Conf.,* 1990, pp 660-666

[15] A. Lioy: Mixed Level Test Generation for MOS Circuits, *European Test Conf.,* 1989, pp 208-211

[16] J. Jacob and N.N. Biswas: GTDB Faults and Lower Bounds on Multiple Fault Coverage of Single Fault Test Sets, *Int. Test Conf.*, 1987, pp 849-855

[17] J.P. Roth: Diagnosis of Automata Failures: A Calculus and a Method, *IBM Journal of Research and Development,* **10**, 1966, pp 278-291

[18] P. Goel: An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits, *IEEE Trans. on Computers,* **C-30** (3), 1981, pp 215-222

[19] L. H. Goldstein: Controllability/Observability Analysis of Digital Circuits, *IEEE Trans. Circuits and Systems,* **CAS-26**, 1979, pp 685-693

[20] J. Sziray: Test Calculation for Logic Networks by Composite Justification, *Digital Processes,* **5**, 1979, pp 3-15

[21] Z. Hegedüs: A Comparative Study about the Efficiency of Digital Test Generation Algorithms, *Ph.D. Thesis,* Technical University of Budapest, 1992 (in Hungarian)

[22] Y. Takamatsu, K. Kinoshita: An Efficient Test Generation Method by 10-V Algorithm, *Int. Symp. on Circuits and Systems,* 1985, pp 679-682

[23] C.W. Cha, W.E. Donath and F. Ozguner: 9-V Algorithm for Test Pattern Generation of Combinational Digital Circuits, *IEEE Trans. Computers,* **C-27** (3), 1978, pp 193-200

[24] Y.H. Levendel and P.R. Menon: Test Generation Algorithms for Computer Hardware Description Languages, *IEEE Trans. Computers,* **C-31**, 1982, pp 577-588

[25] K. Tilly: CONTEST: An Automatic Test Pattern Generation System Based on Constraints, *Technical Report,* Technical University of Budapest, TUB-TR-94-EE11, 1994

[26] K. Tilly: Constraint-Based Automatic Test Pattern Generation, *Ph.D. Thesis,* Hungarian Academy of Sciences, 1995

[27] K. Tilly: A Comparative Study of Automatic Test Pattern Generation and Constraint Satisfaction Methods, *Technical Report,* Technical University of Budapest, TUB-TR-94-EE10, 1994

[28] Steingart, Nagle and Grason: RTG: Automatic Register Level Test Generator, *Proc. IEEE 22nd Design Automation Conference,* 1995, pp 803-807

[29] J. Sziray: The Test-Design Program System DIAS, *Proc. 1st Hungarian Custom Design Conference*, Gyöngyös, May 1987, pp 303-309

[30] J. Sziray and Zs. Nagy: OPART: A Hardware Description Language for Test Generation, *Microprocessing and Microprogramming 32,* North Holland Publishing Company, Amsterdam, 1991, pp 525-530

[31] S.M. Thatte and J.A. Abraham: Test Generation for Microprocessors, *IEEE Trans. on Computers,* **C-29** (6), 1980, pp 429-441

[32] D. Brahme and J.A. Abraham: Functional Testing of Microprocessors, *IEEE Trans. on Computers,* **C-33** (6), 1984, pp 475-485

[33] S.Y.H. Su and Y. Hsieh: Testing Functional Faults in Digital Systems Described by Register Transfer Language, *Proc. IEEE International Test Conf.,* October 1981, pp 447-457

[34] T. Lin and S.Y.H. Su: The S-Algorithm: A Promising Approach for Systematic Functional Test Generation, *IEEE Trans. on Computer-Aided Design,* **CAD-4**, 1985, pp 250-263

[35] S.B. Akers: Functional Testing with Binary Decision Diagrams, *Proc. 8th Int. Symp. on Fault Tolerant Computing,* 1978, pp 82-92

[36] R. Ubar: Test Synthesis with Alternative Graphs, *IEEE Design and Test of Computers,* Spring, 1996, pp 48-59.

[37] R.Ubar and M.Brik: Multi-Level Test Generation and Fault Diagnosis for Finite State Machines. *European Dependable Computing Conf. EDCC-2,* Springer-Verlag, 1996, pp 264-281.

[38] M.D. O'Neill, D.D. Jani, C.H. Cho and J.R. Armstrong: BTG: A Behavioral Test Generator, Computer Hardware Description Languages and their Applications, North Holland, 1990, pp 247-361

[39] C.H. Cho and J.R. Armstrong: VHDL Semantics for Behavioral Test Generation, *Proc. 10th Int. Symp. on Computer HDLs and Their Application,* April 1991, pp 395-412

[40] C.H. Cho and J.R. Armstrong: B-Algorithm: A Behavioral Test Generation Algorithm, *Proc. Int. Test Conf.,* 1994, pp 968-979

[41] B. Benyó: Test Pattern Generation Based on High-Level Hardware Descriptions, *Ph.D. Thesis,* Dept. Measurement and Information Systems, Technical University of Budapest, 1997

[42] Gy. Csertán: A Framework for Early Testability Assessment, *Ph.D. Thesis,* Dept. Measurement and Information Systems, Technical University of Budapest, 1997

[43] M. Abramovici: Digital Sytems Testing and Testable Design, Computer Sci. Press, 1990

[44] N. Gouders: Methoden zur deterministischen Testgenerierung für synchrone Schaltwerke, *Ph.D. Thesis*, Fachbereich Elektrotechnik der Universität - GesamtHochschule - Duisburg, 1991 (in German)

[45] Y. Zorian and M. Marzouki: Multi-Chip Modules Testing and DFT, *38th IEEE Midwest Symposium on Circuits and Systems,* Rio de Janeiro, 1995, pp 722-725.

[46] A. Pataricza, Gy. Csertń, I. Majzik, B. Benyø', B. Sallay and A. Petri: Verification of ultra-reliable controllers. *Internal Report,* Technical University of Budapest, 1997.

[47] A.K. Mackworth: Consistency in networks or relations, *Artificial Intelligence,* **8** (1), 1977, pp 88-118.

[48] U. Montanari: Fundamental properties and applications to picture processing, *Inf. Sci.,* **7**, 1974, pp 95-132

[49] U. Montanari: Optimization Methods in Image Processing *Proc. IFIP Congress,* 1974, pp 727-732

[50] A.K. Mackworth and E.C. Freuder: The complexity of some polynomial network consistency algorithms for constraint satisfaction problems, *Artificial Intelligence* **25** (1), 1984, pp 65-74

[51] E.C. Freuder: A sufficient condition for backtrack-free search, *J. ACM* **29** (1), 1982, pp 24-32

[52] R. Dechter and J. Pearl: Network-based heuristics for constraint satisfaction problems, *Artificial Intelligence,* **34**, 1987, pp 1-38

[53] R. Dechter: Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition, *Artificial Intelligence,* **41**, 1989/90, pp 273-312

[54] D.L. Waltz: Generating semantic descripions from drawings of scenes with shadows, Rep. MAC AI-TR-271 MIT, Cambridge, MA, 1972

[55] J.R. Ullman: Associating parts of patterns, *Information and Control,* **9** (6), 1966, pp 583-601

[56] A. Rosenfeld, R. Hummel and S. Zucker: Scene labeling by relaxation operators, *IEEE Trans. Systems, Man and Cybernetics,* **SMC-6**, 1976, pp 420-433

[57] J. Gaschnig: A general backtrack algorithm that eliminates most redundant tests, *Proc. International Conference on Artificial Intelligence*, Cambridge, MA, 1977, p 457

[58] J. Gaschnig: Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisfying assignment problems, *Proc. 2nd National Conference of the Canadian Society for Computational Studies of Intelligence*, Toronto, July 1978

[59] R.M. Haralick and G.L. Elliott: Increasing tree search efficiency for constraint satisfaction problems, *Artifical Intelligence,* **14**, 1980, pp 263-313

[60] J. Gaschnig: Performance measurement and analysis of certain search algorithms, *Technical Report* CMU-CS-79-124, Carnegie-Mellon University, Pittsburgh, PA, 1979

[61] R.M. Stallman and G.J. Sussman: Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis, *Artificial Intelligence* **9**, 1977, pp 135-196

[62] M. Bruynooghe: Solving combinatorial search problems by intelligent backtracking, *Inf. Process. Lett.* **12**, 1981, pp 36-39

[63] S. Rangajaran, D. Fussell, and M. Malek: Built-in testing of integrated circuit wafers, *IEEE Trans. Comput.*, **39**, 1990, pp 195-205

[64] F.P. Preparata, G. Metze, and R.T. Chien: On the connection assignment problem of diagnosable systems, *IEEE Trans. Electron. Comput.*, **EC-16**, 1967, pp 848-854

[65] F. Barsi, F. Grandoni, and P. Maestrini: A Theory of Diagnosability of Digital Systems, *IEEE Trans. Computers,* **C-25**, 1976, pp 585-593

[66] M. Malek: A comparison connection assignment for diagnosis of multiprocessor systems, *7th Symposium on Computer Architecture,* La Baule, France, May 1980, pp 31-35

[67] K.Y. Chwa and S.L. Hakimi: Schemes for fault-tolerant computing: a comparison of modulary redundant and t-diagnosable systems, *Information and Controls*, **45**, 1981 (3), pp 212-238

[68] J. Maeng and M. Malek: A comparison assignment for self-diagnosis of multicomputer systems, *Proc. 11th FTCS*, pp 173-175, 1981

[69] S.L. Hakimi and A.T. Amin: Characterization of connection assignment of diagnosable systems, *IEEE Trans. Computers*, **C-23**, 1974, pp 86-87

[70] E.R. Scheinerman: Almost sure fault tolerance in random graphs, *SIAM J. Comput.*, **16**, 1987, pp 1124-1134

[71] K. Huang, V.K. Agarwal, L. LaForge, and K. Thulasiraman: A diagnosis algorithm for constant degree structures and its application to VLSI circuit testing," *IEEE Trans. Parallel and Distr. Systems*, 1995, pp 363-372

[72] A.K. Somani and V.K. Agarwal: Distributed diagnosis algorithms for regular interconnected structures, *IEEE Trans. Computers*, **C-41**, 1992, pp 899-906

[73] L.E. LaForge, K. Huang, and V.K. Agarwal: Almost sure diagnosis of almost every good element *IEEE Trans. Computers*, **C-43**, 1994, pp 295-305

[74] P. Maestrini and P. Santi: Self Diagnosis of Processor Arrays Using a Comparison Model, *Proc. 14th Symposium on Reliable Distributed Systems*, pp 218-228, Sept. 1995

[75] S. Chessa: Self-Diagnosis of Grid-Interconnected Systems, with Application to Self-Test of VLSI Wafers, *Ph.D. Thesis,* TD-2/99, Dipartimento di Informatica, Università degli Studi di Pisa, 1999

Publications

[1] B. Sallay, P. Maestrini and P. Santi: Comparison-Based Wafer-Scale Diagnosis Tolerating Comparator Faults, *IEE Journal on Computers and Digital Techniques,* **146** (4), 1999, pp 211-215

[2] S. Chessa, B. Sallay and P. Maestrini: Diagnostic Model and Diagnosis Algorithm of a SIMD Computer, *Proc. 3rd European Dependable Computing Conf.,* Prague, September 1999

[3] B. Sallay: Heuristic Control in the Type-Uninterpreted Dynamic Analysis Phase of Architectural Test Pattern Generation, *Proc. 9th European Workshop on Dependable Computing,* Gdansk, May 1998

[4] B. Sallay: Heuristic Type-Uninterpreted Acceleration of High Level Systematic Test Pattern Generation, *Proc. IEEE European Test Workshop,* Cagliari, May 1997

[5] B. Sallay, A. Petri, K. Tilly and A. Pataricza: Constraint Based High Level Test Pattern Generation for VHDL Circuits, *Proc. IEEE European Test Workshop,* Montpellier, June 1996

[6] K.Tilly, A. Pataricza and B.Sallay: High Level Functional Test Generation Based on Constraint Satisfaction Methods and Heuristic Cost Functions, *2nd Workshop on Hierarchical Test Generation,* Duisburg, September 1995

[7] V. Sieh, A. Pataricza, B. Sallay, W. Hohl, J. Honig, and B. Benyó: Fault Injection Based Validation of Fault Tolerant Multiprocessors, *8th Symp. on Microcomputer and Microprocessor Applications,* Budapest, 1994, pp. 85-94

[8] B. Sallay and A. Petri: On the Heuristic Acceleration of VHDL Based Test Generation Algorithms, *Conf. on the Latest Results of Information Technology,* Budapest, May 1997

[9] A. Petri and B. Sallay: Constraint Based Functional Test Pattern Generation for VHDL Circuit Descriptions, *Int. Conf. of Ph.D. Students,* Miskolc, 11-17 August 1997

[10] B. Sallay, P. Maestrini and P. Santi: A Comparison-Based Diagnosis Algorithm Tolerating Comparator Faults, *Technical Report,* IEI:B4-26-10-98, Istituto di Elaborazione della Informazione, Pisa, Italy,

[11] B. Sallay, B. Benyó, Z. Hegedüs, A. Pataricza, A. Petri, J. Sziray, K. Tilly: A Proposed VHDL Subset for Test Generation Purposes, *Technical Report,* FUTEG-1/1994

[12] B. Sallay, K. Tilly, A. Pataricza, Z. Hegedüs, A. Petri, L. Surján and J. Sziray: An Artificial Intelligence Based Approach to VHDL Level Test Pattern Generation, *Technical Report,* FUTEG-2/1994

[13] B. Sallay, K. Tilly, A. Pataricza, Gy. Csertán, Z. Hegedüs, A. Petri, L. Surján and J. Sziray: Application of AI Methods in High Level Test Generation, *Technical Report,* FUTEG-3/1995

[14] B. Sallay, K. Tilly, A. Petri, A. Pataricza and J. Sziray: AI Methods in High Level Test Generation - A Feasibility Study, *Technical Report,* FUTEG-4/1995

[15] A. Pataricza, Gy. Csertń, I. Majzik, B. Benyø', B. Sallay and A. Petri: Verification of Ultra-Reliable Controllers. *Internal Report,* Technical University of Budapest, 1997.

Appendix A

Benchmark circuits

A.1 Gate-level circuits

A.1.1 Adder family

This is the classical scalable carry-chain architecture of an adder unit. It contains a chain of *full adder* components (Figure A.1) which implement 1-bit addition. The full adder component are cascaded to perform multiple-bit addition, as shown in Figure A.2. Characteristic data of the family are given in Table A.1.

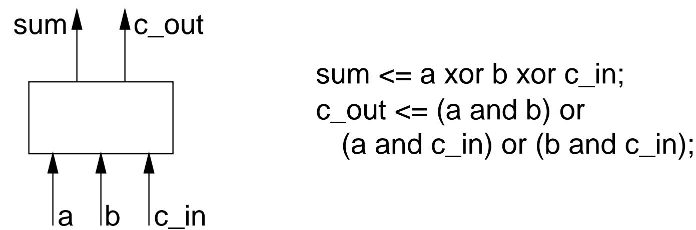

Figure A.1: Full adder

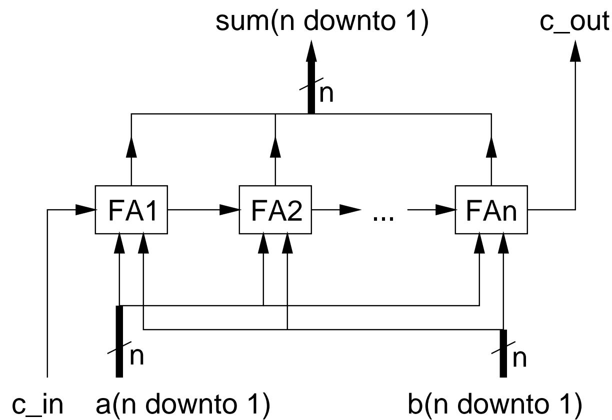

Figure A.2: n-bit adder

circuit	signals	inputs	outputs	components	detectable ratio
full adder	15	3	2	12	100%
4-bit adder (hierarchical)	15	3	2	4	100%
4-bit adder (flattened)	57	3	2	48	100%
8-bit adder (hierarchical)	33	3	2	8	100%
8-bit adder (flattened)	113	3	2	96	100%

Table A.1: The adder family

A.1.2 ISCAS'85 family

The ISCAS'85 benchmark set [11] contains ten large gate-level structures. The quantitative features of this set is reported in Table A.2. Since it is a gate-level benchmark, the used component set is the fix {*inverter, and-gate, or-gate, nand-gate, nor-gate, xor-gate, buffer*}.

circuit	signals	inputs	outputs	gates
c432	196	36	7	160
c499	243	41	32	202
c880	443	60	26	383
c1355	587	41	32	546
c1908	913	33	25	80
c2670	1426	233	140	1193
c3540	1719	50	22	1669
c5315	2485	178	123	2307
c6228	2448	32	32	2416
c7552	3719	207	108	3512

Table A.2: The ISCAS'85 family

A.2 High-level benchmarks

The benchmark circuits are considered as architectural if the following features appear in their description:

- They contain high-level components.
- Multiple-bit data are handled together by components.
- There is some separation between data and control.

Since we deal with abstract elements, we also give the number of bit-wide signals and gates a surface-optimised synthesizer tool would generate.

A.2.1 Combinational multiplier

This example is taken from the Mentor tool [8] tutorial library. The architecture implements the well-known *shift-and-add* multiplication technique. Figure A.3 shows the 4-bit implementation of this circuit. Constant-offset shifters are implemented as the addition of grounded wires. A good high-level synthesizer tool (see Section 2.2.2) can be directed to generate the architecture of Figure A.3 by the VHDL code below. Of course, the "*" and "+" operators must be properly overloaded to implement 1-bit multiplication between vector and bit, and addition between two vectors, respectively.

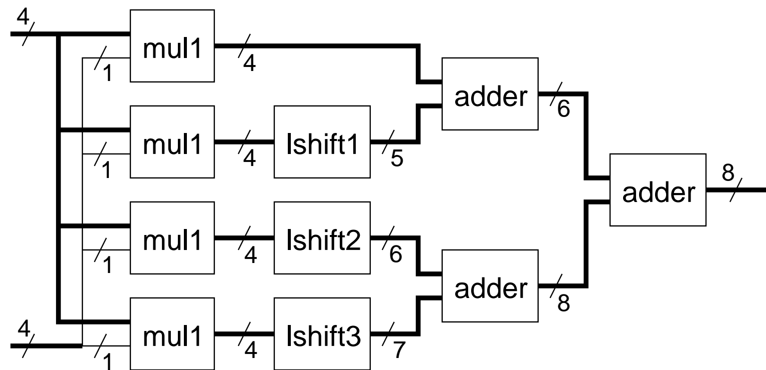

Figure A.3: 4-bit combinational multiplier

```
res <= ( in1*in2(0) + (in1*in2(1))&'0' ) +
       ( (in1*in2(2))+"00" + (in1*in2(3))&"000" );
```

Multiplier circuit data are summarised in Table A.3.

circuit	signals	domain size	inputs	outputs	components	gate equivalent	detectable ratio
2-bit multiplier	5	13	2	1	3	40	96.15%
4-bit multiplier	9	51	2	1	7	256	94.11%
8-bit multiplier	17	199	2	1	15	1168	92.96%

Table A.3: The multiplier family

A.2.2 Greatest common divisor

The GCD circuit is a high-level synthesis benchmark circuit because it provides opportunity for optimisations. The design uses Euler's algorithm to find the largest number which divides both the operands:

```
while op1 /= op2 loop
   if op1 < op2 then
      op2 := op2 - op1;
   else
      op1 := op1 - op2;
   end if;
end loop;
```

We have chosen one of the many possibilities this circuit can be synthesized (Figure A.4).

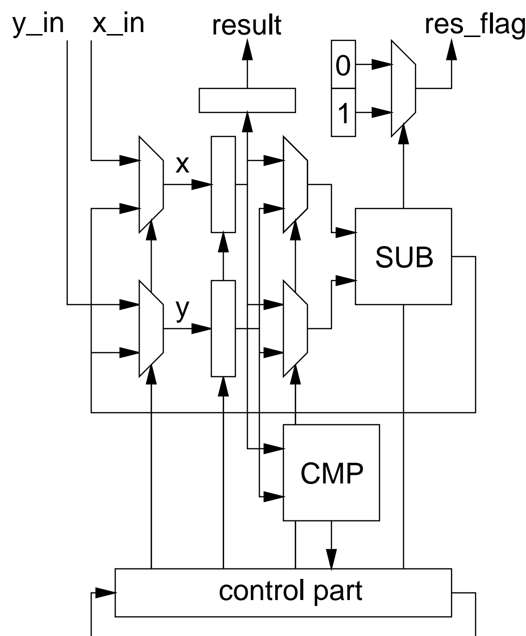

Figure A.4: Greatest common divisor

GCD is a highly data-intensive circuit where control is also signifant, although the control part has a small number of states. The control part has been synthesized to the gate-level. The benchmark is present in the subset with different word sizes for the data signals. Table A.4 shows a summary on the characteristics of different gcd instances. GCD is a true high-level benchmark because the number of signals does not grow with the word length and because there are abstract integer operations. Since many physical faults require 3, 4, or 5 time frame long test sequences, the total domain size which must be explored is multiplied accordingly. The ratio of detectable faults is given for at most 3-long test sequences.

It is obviously not worth choosing this implementation for a 2-bit word length where a two-level combinational circuit would be much faster and smaller, but our purpose is the evaluation of the ATPG algorithm for different versions of the same circuit.

circuit	signals	domain size	inputs	outputs	components	gate equivalent	detectable ratio
2-bit gcd	21	32	2	1	17	107	64.06%
4-bit gcd	21	52	2	1	17	203	66.34%
8-bit gcd	21	92	2	1	17	395	67.93%

Table A.4: The GCD family

A.2.3 Bubble sort

The bubble sort circuit implements the well-known sorting algorithm where adjacent containers exchange values in case their order does not correspond to the desired one. Although there are many control signals again, the gate-level control part has few states: in addition to the initial register load state, it contains an *even* phase when even-indexed registers exchange values with their right neighbours, and an *odd* state when odd-indexed registers do the same. The architectural model implementing this algorithm is shown in Figure A.5. Although the exact structure is not important, it can be seen that a lot of multiplexers organize the feedback paths. The control part is fed by comparators. (Multiplexer control signals are not shown.) The bubble sort circuit exists again in several intances of different word sizes (Table A.5). The detectable coverage is given for 3-time frame long tests.

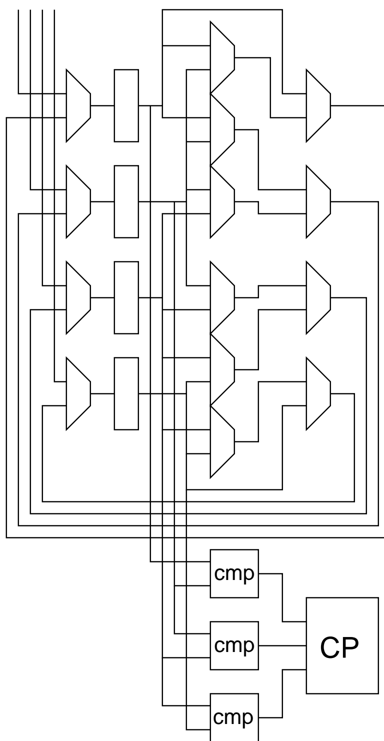

Figure A.5: Bubble sort

circuit	signals	domain size	inputs	outputs	components	gate equivalent	detectable ratio
2-bit BS	42	68	4	1	32	194	52.94%
4-bit BS	42	114	4	1	32	364	54.38%
8-bit BS	42	206	4	1	32	704	55.34%

Table A.5: The bubble sort family

Appendix B

The BudaTest program

BudaTest is a library-based architectural TPG program written in C++. The source code of the program is about 5000 lines of which about 1300 belongs to the experimental but extendable constraint library. BudaTest is a portable command line-driven tool and compiles well under UNIX and Windows platforms. There exists an X-windowing front-end, developed by András Petri, for the UNIX-based BudaTest version.

The block diagram of the tool is shown in Figure B.1. The tool contains three parsers: a VHDL and an ISCAS input description parsers, and a component test description parser for hierarchical test generation. The supported VHDL subset is structural VHDL. The parsers have been created using the *lex* and *yacc* compiler development tools.

The circuit model contains signal type information and signal-component interconnection information. The default fault list contains the two sa-faults for every signal bit, with or without the use of component test files. For every fault, the program creates the CSP by mapping the VHDL signals into constraint variable pairs and the components into constraint library elements. Since sequential input circuits are allowed, many CSP elements may belong to the same VHDL object. The CSP engine implements all the techniques dexcribed in Section 6. Since BudaTest does not contain yet well-adjusted heuristics, the heuristic engine currently contains only a few heuristic order definitions that prevent the solver from the totally random selection of assigned variables and values.

BudaTest is invoked the following way:

```
> budatest
BudaTest architectural test pattern generator
  by Balazs Sallay, Technical University of Budapest
  version 3.07

Usage:
  budatest <switches> -input <ifile> [-output <ofile>] [-max <framenr>]
     [-timeout <tnr>] [-fault <fsignal> <fvalue> [<fpos>]]
```

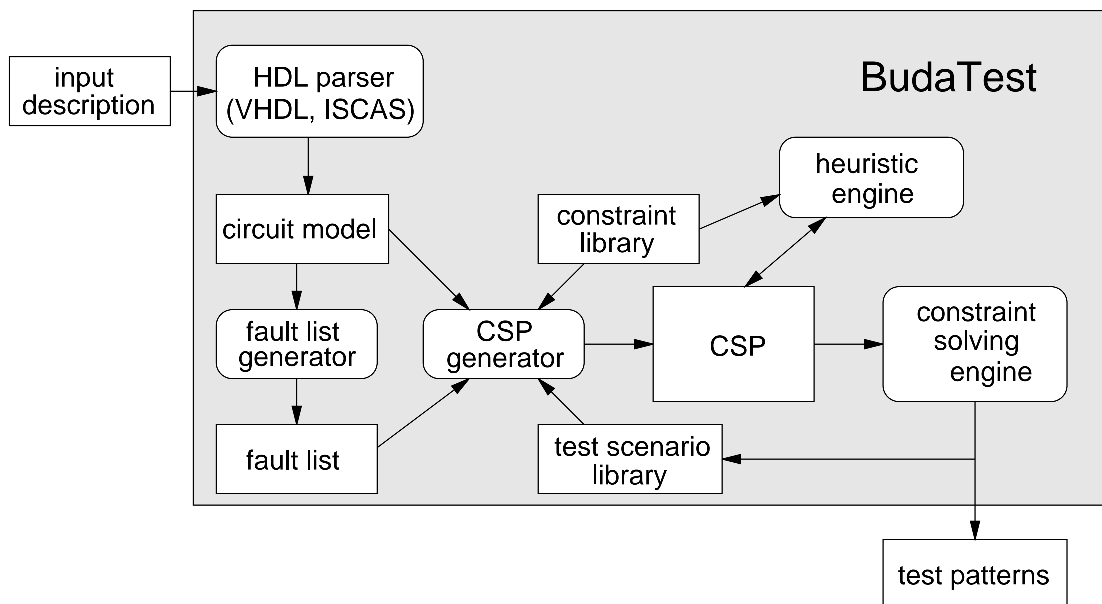

Figure B.1: BudaTest block diagram

```
ifile: file name of the VHDL description of the circuit
ofile: output component test file name
framenr: number of sequential frames for sequential circuits
tnr: limit decision tree to <tnr>*2^16 nodes, then give up
fsignal: VHDL identifier of the faulty signal
fvalue: stuck-at value of the fault
fpos: bit position of the stuck-at fault
-iscas: use ISCAS parser instead of VHDL
-parsedebug: debug VHDL or ISCAS parser
-all: generate and test all faults
-component: look for component test files
-random: random order of constraint assignments
-nologic: disable mask and interval logic
-noimpl: disable implication enhancement
-nonode: disable node classificatiopn enhancement (implies -nocolour)
-nocolour: disable colouring enhancement
```

The program options reflect our discussion of the modelling and evaluation techniques in Sections 5 and 6.