J Electron Test (2017) 33:751–767
https://doi.org/10.1007/s10836-017-5693-6

Evaluating the Effectiveness of D-chains in SAT-based ATPG and Diagnostic TPG

Pascal Raiola[1] · **Jan Burchard**[1] · **Felix Neubauer**[1] · **Dominik Erb**[2] · **Bernd Becker**[1]

Received: 9 August 2017 / Accepted: 10 November 2017 / Published online: 8 December 2017

Abstract The ever increasing size and complexity of today's Very-Large-Scale-Integration (VLSI) designs requires a thorough investigation of new approaches for the generation of test patterns for both test and diagnosis of faults. SAT-based automatic test pattern generation (ATPG) is one of the most popular methods, where, in contrast to classical structural ATPG methods, first a mathematical representation of the problem in form of a Boolean formula is generated, which is then evaluated by a specialized solver. If the considered fault is testable, the solver will return a satisfying assignment, from which a test pattern can be extracted; otherwise no such assignment can exist. In order to speed up test pattern generation, the concept of D-chains was introduced by several researchers. Thereby supplementary clauses are added to the Boolean formula, reducing the search space and guiding the solver toward the solution. In the past, different variants of D-chains have been developed, such as the backward D-chain or the indirect D-chain. In this work we perform a thorough analysis and evaluation of the D-chain variants for test pattern generation and also analyze the impact of different D-chain encodings on diagnostic test pattern generation. Our experimental results show that depending on the incorporated D-chain the runtime can be reduced tremendously.

Keywords Automatic test pattern generation · D-chain · SAT-based ATPG · Test pattern generation for diagnosis

1 Introduction

With the increasing size of today's VLSI designs, classical automatic test pattern generation (ATPG) algorithms as used in state-of-the-art commercial tools start to run into scalability issues. Thus, test pattern generation – even for standard stuck-at tests – may require several weeks on a whole server-farm in order to guarantee a high fault coverage and a compact test set. Consequently, new algorithms are required that may reduce the runtime significantly. In this context, SAT-based algorithms are getting more and more popular [4, 5, 7, 10, 11, 13, 20, 25, 27, 31, 32, 34, 37]. These algorithms promise a remarkable performance even on large industrial benchmarks [29, 34] and are clearly superior compared to classical algorithms as soon as faults are processed for which no test pattern exists. Furthermore, they support higher valued logics [13, 14, 27], waveform accurate encoding of timing information [24] and integration of sophisticated fault models [4, 11, 25, 37] in a convenient way. Algorithms to solve *Pseudo-Boolean Optimization* (PBO) problems [1] are often built on top

Responsible Editor: L. M. Bolzani Pöhls

✉ Pascal Raiola
raiolap@informatik.uni-freiburg.de

Jan Burchard
burchard@informatik.uni-freiburg.de

Felix Neubauer
neubauef@informatik.uni-freiburg.de

Dominik Erb
dominik.erb@infineon.com

Bernd Becker
becker@informatik.uni-freiburg.de

[1] Computer Architecture, University of Freiburg, Freiburg (Breisgau), Germany

[2] Infineon Technologies AG, Neubiberg, Germany

of SAT-algorithms. Using so-called PBO-SAT allows the handling of non-trivial constraints and as such is also applicable to ATPG for advanced fault models [9].

In contrast to classical structural ATPG, a SAT-based algorithm first generates a mathematical representation of the problem. Afterwards, the resulting equation is evaluated by a specialized solver to determine the testability of a fault and to extract a test pattern in case a satisfying assignment was found. The concept of D-chains, which extend the mathematical model with additional information, was introduced [20, 31] to increase the solving speed. D-chains drastically reduce the search space by forcing the solver to only consider assignments that could lead to a valid test pattern. With the advent of incremental solving [8] new concepts like backward D-chains [12] or indirect D-chains [2, 5] were introduced.

In [2] we furthermore analyzed different D-chain concepts for test pattern generation and evaluated their benefits depending on the considered fault model and circuit type for the first time. In this work we elaborate on the investigation of [2] and expand it by an analysis on the impact of different D-chain concepts on SAT-based *diagnostic* test pattern generation. In particular, we present:

- A thorough investigation of the different D-chain concepts that have been proposed so far to evaluate which is the best method for different kinds of problems for ATPG.
- An indirect D-chain algorithm with two **novel hybrid** extensions that not only achieves good overall speedups but is especially well suited for cryptographic circuits.
- Experimental results for ATPG considering stuck-at and transition-delay faults, for different kinds of academic and industrial benchmarks.
- An investigation on diagnostic test pattern generation (DTPG), where the new concept of the *Early Target Backward Implication D-chain* is introduced and experimental results for academic and industrial benchmarks are presented.

Our experimental results show that depending on the incorporated D-chain the total solve time can be significantly reduced by over 90% compared to an algorithm that does not include a D-chain. Furthermore, the notable advantages of the backward, indirect and hybrid D-chain approaches compared to the standard forward D-chain are demonstrated.

The rest of the paper is organized as follows: Section 2 introduces the fault models as well as SAT-based ATPG and DTPG. Afterwards, in Section 3 we present the different D-chain techniques for ATPG and introduce our new hybrid indirect D-chains. In Section 4 we transfer the D-chains to DTPG and propose the new early target backward D-chain. Subsequently, we evaluate the different D-chains in Section 5 and conclude with a short summary and outlook in Section 6.

2 Preliminaries

2.1 Automatic Test Pattern Generation

In the area of circuit test, fault models are widely used to abstract from real physical defects toward a formal model. Based on such a fault model, the circuit can be tested for the presence or absence of the modeled faults using input stimuli, so-called *test patterns*.

Automatic test pattern generation (ATPG) algorithms compute such test patterns for a given fault if they exist and are utilized throughout the industry. Historically, structural methods like the D-algorithm [22] or its improvements [15, 18] proved to be successful in generating test patterns and have been studied in great detail. A newer technique is SAT-based ATPG which leverages the ever increasing power of Boolean satisfiability (SAT) solvers to generate test patterns. Unlike structural methods which work directly on the circuit, methods based on SAT convert the entire ATPG problem into a mathematical representation, a Boolean formula. This formula is then evaluated by a SAT solver and a test pattern can be extracted from the satisfying variable assignment. If there is no satisfying assignment (the formula is *unsatisfiable*), it is proven that no test pattern for the fault exists.

2.2 Fault Models

In this article, we focus on the two widely used fault models *stuck-at* [16] and *transition-delay* [3, 19] which can both be easily handled by a SAT-based ATPG. In more complex fault models the difficulty of a successful fault propagation or justification might be much larger and D-chains even more important [12, 13]. Generally, the pattern generation for most fault models can be transformed to the SAT problem with ease and efficiently solved with ever more powerful SAT-solvers, which is one of the great benefits of this approach.

In the (single) stuck-at fault model a single line in the circuit is always ‘0’ or ‘1’ (referred to as *stuck-at-0* and *stuck-at-1*, respectively) independent of the line’s true value. The transition-delay model further refines these strict conditions by instead considering *slow-to-rise* and *slow-to-fall* faults evolving over two time frames. A line with a slow-to-rise fault that is ‘0’ in the first time frame will maintain this ‘0’ even if it switches to ‘1’ in the second time frame in a fault-free circuit. Similarly, a line with a slow-to-fall fault maintains a ‘1’ of the first time frame even if it switches to ‘0’ in the second one. Hence, the transition-delay fault model also covers defects where lines are not stuck at a fixed value but also do not react to value changes at the required speed. Not all such defects can be detected with the stuck-at model. However, the transition-delay model comes with the additional cost of modeling two time frames instead of only one.

2.3 Conversion to SAT

The general conversion of a circuit into a Boolean formula is well studied and mainly consists of introducing variables for the circuit inputs and every gate output and subsequently applying the Tseitin transformation [35] to every gate. The resulting formula represents the circuit and a satisfying variable assignment corresponds to an input assignment to the circuit as well as a valid propagation of this assignment. It should be noted that our approach is capable of directly mapping standard gates with more than two inputs into CNF as well. This offers a much more efficient encoding than a decomposition of these gates into multiple two-input gates. Complex cells (e.g., AND-OR-Invert cells) are mapped to standard gates based on their description in the cell library.

Based on this conversion, a SAT-based ATPG algorithm utilizes two representations of the circuit: A fault-free version is used to evaluate the circuit under normal conditions and a fault-affected version tracks the influence of the current fault. If an output of the two versions differs for the same input assignment, the fault is detected and the input assignment can be used as a test pattern. The difference between outputs is encoded through one XOR gate per output, resulting in a miter circuit (see Fig. 1) which is transformed into a Boolean formula. In addition, the variables representing the outputs of the XOR clauses are combined into a single clause. This ensures that the formula is only satisfiable if at least one of the outputs shows a difference.

2.3.1 Cones of Influence

For efficiency, only the required parts of the circuit are transformed into the formula. These parts are marked by a cone of influence computation (see Fig. 2) which greatly reduces the overall size of the formula.

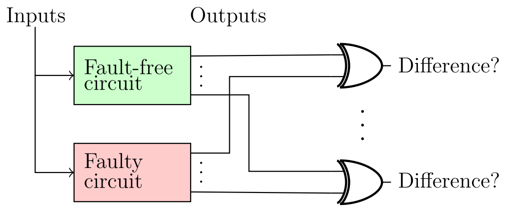

Fig. 1 A miter circuit consisting of the fault-free and fault-affected circuit and an XOR gate for each output

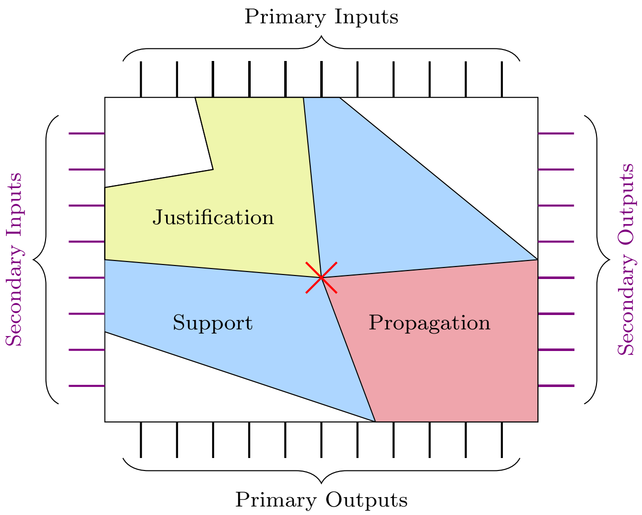

Fig. 2 The cones of influence which need to be modeled

The justification cone of the fault site and the support for the fault propagation are only required once since they are independent of the fault. Thus, only the propagation cone needs to be modeled in two different versions representing the fault-free and faulty circuit. Here, signals are represented not only by a G variable (for the *good* circuit) but also an additional B variable for the *bad* version.

For the transition-delay fault model the first time frame needs to be modeled as well. In this time frame the faulty line has to be charged to the required value. In addition, assuming a launch-on-capture test architecture [26], all flip-flop values required in the second time frame need to be set in the first time frame. This requires additional cones of influence, but the overall modeling remains unchanged.

2.3.2 Modeling the Fault

For the stuck-at fault model, the fault-affected line is simply cut into two parts (represented by two variables): The second part is forced to the value that the line is stuck at. The first part of the line is forced to the inverse of the stuck-at fault (to ‘1’ for a stuck-at-0 and vice-versa) which ensures the fault activation.

Similarly, the line affected by a transition-delay fault is also split into two parts in both time frames. The first part is forced to the required transition (e.g., for a slow-to-rise fault to ‘0’ in the first time frame and to ‘1’ in the second), whereas the second part behaves like a stuck-at fault (e.g, for a slow-to-rise fault the line will stay at ‘0’ in both time frames).

2.4 Incremental Solving

To be able to detect a fault, its effect has to be visible on at least one output. The incremental solving approach [33] first generates a formula which is satisfiable if the fault effect can be seen at the first output in the propagation cone of the fault. This formula is usually much smaller because it only contains the gates in the input cone of the modeled output. If the formula is satisfied, the fault is detected and the ATPG can continue with the next fault. Otherwise the formula is extended with clauses representing the input cone of the second output, reusing all previously modeled gates. This approach continues until a test pattern was found or all outputs have been tried, in which case the fault is not detectable.

Incremental solving provides the benefits of initially smaller formulas and of guiding the search process because the fault effect has to be propagated to a single output. Therefore, especially for easy to solve instances it provides large increases in solving speed.

2.5 Diagnostic Test Pattern Generation (DTPG)

In contrast to fault detection, the objective of fault diagnosis is to investigate the location of the fault. Faults at different locations can be distinguished by applying a test pattern that causes different output signal values for the given faults – such a test pattern is called a diagnostic test pattern. We implemented a SAT-based approach to generate diagnostic test patterns with the goal of distinguishing all faults, that can be distinguished.

For the generation of such a diagnostic test pattern, a miter circuit (see Fig. 3) similar to the miter circuit for ATPG is utilized and transformed into a Boolean formula. Note that in contrast to the miter for the generation of a fault detection test pattern, here a difference between two faulty circuit versions is considered. The XOR gates encode an output difference between the respective propagated fault effect, thus the corresponding Boolean formula is satisfiable if a pattern exists which allows the detection of only one fault at least at one output.

Similar to the encoding described in Section 2.3.1, we only encode the required parts of the circuit. However, in the context of DTPG two faults with their respective cones need to be encoded (see Fig. 4).

In general, a diagnostic test pattern generation algorithm uses classification to distinguish every distinguishable fault-pair. It hereby starts with one class containing all faults, picks two faults out of that class and aims at generating a diagnostic test pattern which distinguishes the two faults. If no such test pattern can be found, the faults are marked as indistinguishable; in case a distinguishing test pattern is found, every fault is simulated once for the given pattern. The respective output values are used to divide the current classes into multiple classes of (yet) indistinguishable faults. Note that indistinguishable faults form equivalence classes for two-valued logic [21, 23], thus an indistinguishable fault pair can be merged for diagnostic classification.

This process is repeated for every class until no class contains any distinguishable faults.

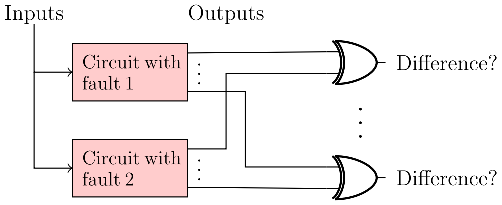

Fig. 3 A miter circuit consisting of two representations of the circuit, each affected by a different fault, and an XOR gate for each output

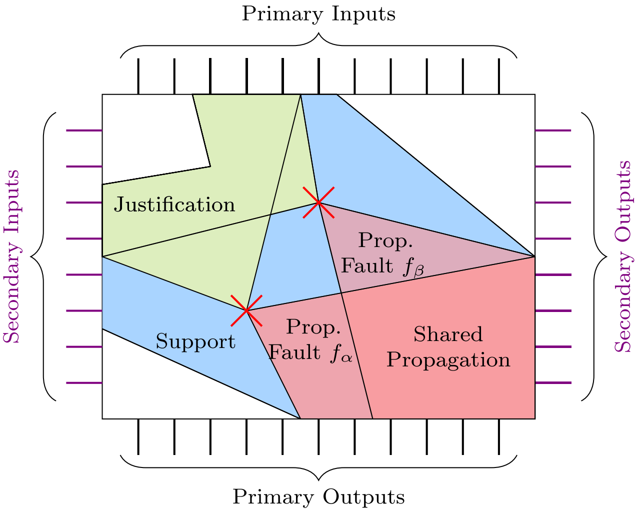

Fig. 4 The cones of influence which need to be modeled for diagnostic test pattern generation. The two faults f_α and f_β are marked with red crosses

3 D-chains in ATPG

The general SAT-based ATPG algorithms introduced in the previous chapter can generate all possible test patterns. However, a drawback of utilizing a SAT solver is that the structural information of the circuit is not directly used by the solver. Hence, it might occur that the solver spends a large fraction of the solve time in regions of the search space which will never lead to a valid test pattern.

D-chains add redundant information based on structural information to the Boolean formula which helps to guide the solve process [20]. This is achieved by augmenting each gate output with a new variable D which encodes whether there is a *difference* between the fault-free and fault-affected version of the circuit.

Examples for such augmentation based D-chains are the forward [20, 31] and backward [12] implication D-chains as well as a combination of the two. In addition we presented in [2] an indirect D-chain [5] implementation and two novel hybrid variants which reduce the amount of redundant information. All of these D-chain types are described in detail in the following subsections.

3.1 Forward Implication D-chain

The forward implication D-chain attempts to enforce the propagation of the difference along paths to an output. For each gate output a new variable D_f is introduced. Assigning D_f to ‘1’ implies that there is a difference at this output:

$$D_f \Rightarrow (G \oplus B) \tag{1}$$

or equivalently in conjunctive normal form (CNF):

$$(\overline{D_f} \vee G \vee B) \wedge (\overline{D_f} \vee \overline{G} \vee \overline{B}) \tag{2}$$

Assuming that the gate output is connected to n successor gates with the difference variables $D_{f1}, \ldots, D_{fn}$ the following D-chain clause is added to the formula:

$$D_f \Rightarrow (D_{f1} \vee D_{f2} \vee \cdots \vee D_{fn}) \tag{3}$$

A difference at the current gate output implies that the difference will propagate to at least one output of a successor gate. In case it is not possible to propagate the difference to any successor output D_f cannot be assigned to ‘1’.

As an example consider the gate Gt_1 in Fig. 5. The gate’s output is connected to two gates and the D-chain clause becomes $D_{f_1} \Rightarrow (D_{f_2} \vee D_{f_3})$ as indicated by the red arrows.

When combined with incremental solving, extra care has to be taken to ensure the forward D-chain implies only D variables of gates that are actually modeled. This is achieved by forcing all D variables of gates which are not modeled but occur in the forward D-chain to ‘0’.

3.2 Backward Implication D-chain

While the forward implication D-chain adds a chain which implies the D_f values of succeeding nodes, the backward D-chain implies the D values of preceding nodes.

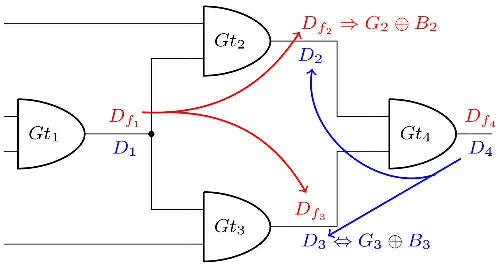

Fig. 5 Circuit with an example for the forward implication D-chain in red and an example for the backward implication D-chain in blue

Assuming that a gate has m inputs with the difference variables $D_1, \ldots, D_m$, its output can only show a difference if at least one of the inputs also shows a difference:

$$D \Rightarrow (D_1 \vee D_2 \vee \cdots \vee D_m) \tag{4}$$

This also allows for a strengthening of the difference variable:

$$D \Leftrightarrow (G \oplus B) \tag{5}$$

Should one of the gate’s inputs be outside the fault propagation cone it cannot have a difference and the corresponding D variable is simply omitted from Formula (4). In the example circuit in Fig. 5, the backward D-chain is indicated in blue and creates the clause $D_4 \Rightarrow (D_2 \vee D_3)$ for the gate Gt_4.

The backward D-chain can be easily implemented in combination with incremental solving, which considers all gates in the input cone of the current output. The entire backward D-chain for this cone can be created because all these gates are guaranteed to be part of the current iteration.

3.3 Combined D-chains

When a backward D-chain is already in place, a cheaper forward D-chain can be added. This is because Formula (1) which requires two clauses can then be replaced by the single clause

$$D_f \Rightarrow D \tag{6}$$

Thus, the combination of both D-chains requires slightly fewer clauses than the sum of the single implementations of both D-chains.

3.4 Indirect D-chain

The previously discussed D-chains add extra information to the formula to guide the solver. In contrast, we present an indirect D-chain which reduces the amount of redundant information by completely removing the B value. Instead, for every signal in the fault propagation cone only the good value G and the difference D are computed. The idea for such an indirect D-chain was first introduced in [5]. However, the presented D-chain is limited to gates with at most two inputs only. The indirect D-chain presented in this work is applicable to all gates, including those with more than two inputs.

While the computation of the G value can be easily performed by converting the gate with the Tseitin transformation, the D value at a gate output cannot be derived so easily. As an example consider Formula (7) which shows the required clauses for a two input AND gate with inputs represented by the variables G_1 and G_2, differences D_1 and D_2, and an output with the difference variable D.

$$\begin{aligned} D \Rightarrow\ &\big((D_1 \vee D_2) \wedge (D_1 \vee G_1) \wedge (D_2 \vee G_2) \\ &\wedge (G_1 \vee \overline{G_2} \vee \overline{D_2}) \wedge (G_2 \vee \overline{G_1} \vee \overline{D_1})\big) \\ \overline{D} \Rightarrow\ &\big((\overline{G_1} \vee D_1 \vee \overline{D_2}) \wedge (\overline{D_1} \vee \overline{G_2} \vee D_2) \\ &\wedge (G_1 \vee \overline{D_1} \vee G_2 \vee \overline{D_2}) \wedge (\overline{G_1} \vee \overline{G_2} \vee \overline{D_2})\big) \end{aligned} \tag{7}$$

Directly deriving the D value instead of first computing B and using an XOR to obtain it, can result in a smaller overall formula. This is especially the case on the outer perimeter of the fault propagation cone where many gates have only one input which can be different. However, the number of clauses grows exponentially with the number of gate inputs potentially showing a difference. Therefore the overall formula size could grow significantly large, depending on the structure of the circuit.

3.5 Hybrid Indirect D-chain

Based on the previous observations, the hybrid D-chain modeling attempts to combine the conventional D-chain concept with the indirect method. Generally, the gates in the fault propagation cone are encoded using the indirect method. However, for gates with a large number of inputs that can potentially show a difference between the fault-free and the faulty circuit, a B value is derived from these inputs ($B \Leftrightarrow G \oplus D$) and the gate is encoded through the Tseitin transformation. The gate output's D value is then re-computed as in Formula (5).

For the selection of gates which are to be encoded in the classical manner we developed two heuristics. They are both based on the number of circuit inputs that can potentially have a difference between the good and bad version of the circuit.

- Static Selection: Models all gates where more than one input can have a difference in a conventional manner. This heuristic is based on the observation that the indirect encoding is especially beneficial when only few circuit inputs can have a difference. When, on the other hand, many gate inputs have a difference, the conventional encoding might be cheaper and more efficient.
- Dynamic Selection: The dynamic selection heuristic extends the static selection based on the observation that any change from indirect to conventional encoding (and back) is rather expensive since additional XOR operations have to be performed. Nodes are selected for conventional modeling in a three step approach. In the first step the *score* of each node is computed. The *score* is the number of inputs that can show a difference. Next, the combined successor score *css* for each node is computed as the sum of the score of all successor nodes. Each node with a *css* of two or larger has to provide a B value at its output. This can be achieved in two ways: Either the node is modeled conventionally or an XOR gate is added to re-create it. In the last step, the final decision regarding the conventional modeling is performed. When a node has more than one input that can have a difference, and for at least all but one of these inputs a B value is available, it will be encoded conventionally.

Both heuristics attempt to strike a balance between the two modeling methods with the hope of resulting in a smaller and more efficient overall formula and faster solving speed.

4 D-chains in Diagnostic TPG

While in the context of automatic test pattern generation a difference between the fault-free and the fault-affected circuit is targeted, a *diagnostic* test pattern is required to produce a difference between two circuits, which are affected by different faults f_α and f_β.

As only the required parts of the circuit are encoded in the Boolean formula (see Fig. 4), the targeted difference at the outputs varies depending on the propagation cone an output is contained in:

- If an output is contained in the propagation cone of solely one fault, the given fault pair can be distinguished, if the fault can be detected at that output.
- If the output is in the propagation cone of both faults, the faults can be distinguished at that output, if only one fault is detected.

The respective targeted differences are given in Table 1.

In the following we present different D-chain concepts for diagnostic test pattern generation, which are derived from the D-chains presented in Section 3. This work furthermore introduces the concept of the *early target* backward implication D-chain, which aims at guiding the solver from the outputs toward a fault-site, while distinguishing two faults. This D-chain is unique to diagnostic pattern generation and cannot be applied to ATPG.

Table 1 Variation of difference encoding for diagnostic TPG

prop. cone f_α	prop. cone f_β	targeted difference
✓	✗	$G \oplus B^\alpha$
✗	✓	$G \oplus B^\beta$
✓	✓	$B^\alpha \oplus B^\beta$

4.1 Forward Implication D-chain

In diagnostic test pattern generation the forward implication D-chain aims at propagating a fault difference – if existent – from a fault site to succeeding nodes, introducing the new variables D_f^α and D_f^β for each gate output in the respective fault cone:

$$D_f^\alpha \Rightarrow (G \oplus B^\alpha) \quad \text{and} \quad D_f^\beta \Rightarrow (G \oplus B^\beta) \tag{8}$$

Thus, depending on which propagation cone a signal line is contained in, it potentially has 5 variables corresponding to the signal (see Table 2).

Assume that $D_{f1}^\alpha, D_{f2}^\alpha, \ldots D_{fn}^\alpha$ and $D_{f1}^\beta, D_{f2}^\beta, \ldots D_{fn}^\beta$ are the difference variables of all n successor gates. Then the following D-chain clauses are created, formalizing, that a fault effect is propagated to at least one output of a successor gate:

$$D_f^\alpha \Rightarrow (D_{f1}^\alpha \vee D_{f2}^\alpha \vee \cdots \vee D_{fn}^\alpha) \tag{9}$$

$$D_f^\beta \Rightarrow (D_{f1}^\beta \vee D_{f2}^\beta \vee \cdots \vee D_{fn}^\beta) \tag{10}$$

A test pattern distinguishes the two faults f_α and f_β if the targeted difference from Table 1 holds for one output. Some targeted differences can be equivalently described on the basis of the D-values, as displayed in Table 3.

4.2 Backward Implication D-chain

Contrary to the forward implication D-chain, the backward implication D-chain targets propagating a fault difference – if existent – to preceding nodes, introducing the new variables D^α and D^β for each gate output in the respective fault cone:

$$D^\alpha \Leftrightarrow (G \oplus B^\alpha) \quad \text{and} \quad D^\beta \Leftrightarrow (G \oplus B^\beta) \tag{11}$$

Thus, for each signal line there are potentially 5 variables corresponding to the signal (see Table 4).

Assume that $D_1^\alpha, D_2^\alpha, \ldots D_m^\alpha$ and $D_1^\beta, D_2^\beta, \ldots D_m^\beta$ are the difference variables of all m predecessor gates. Similar to the creation of Eqs. 9 and 10, the following D-chain clauses are generated, formalizing, that a gate output can only show a fault effect, if a fault effect shows at least at one of the gate's inputs:

$$D^\alpha \Rightarrow (D_1^\alpha \vee D_2^\alpha \vee \cdots \vee D_m^\alpha) \tag{12}$$

$$D^\beta \Rightarrow (D_1^\beta \vee D_2^\beta \vee \cdots \vee D_m^\beta) \tag{13}$$

Table 2 Values used for the *forward implication D-chain* in diagnostic test pattern generation

prop. cone f_α	prop. cone f_β	encoded signals
✗	✗	G
✓	✗	G, B^α, D_f^α
✗	✓	G, B^β, D_f^β
✓	✓	$G, B^\alpha, B^\beta, D_f^\alpha, D_f^\beta$

Table 3 Variation of difference encoding for diagnostic TPG

prop. cone f_α	prop. cone f_β	targeted difference
✓	✗	D_f^α
✗	✓	D_f^β
✓	✓	$B^\alpha \oplus B^\beta$

The targeted differences can be equivalently described on the basis of the D-values, as displayed in Table 5. Note that all targeted differences are expressed by using solely D-Literals.

4.3 Early Target Backward Implication D-chain

As an alternative to checking the targeted difference only at the outputs, a variable D^T can be introduced for each gate g. The definition of D^T depends on which fault propagation cone contains g, similarly as for the targeted difference in Table 1:

$$D^T := \begin{cases} B^\alpha \oplus B^\beta & \text{if both prop. cones contain } g \\ G \oplus B^\alpha & \text{if prop. cone of } f_\alpha \text{ contains } g \\ G \oplus B^\beta & \text{if prop. cone of } f_\beta \text{ contains } g \end{cases} \tag{14}$$

Then there is no need to separately compute the fault differences, as it was the case for Eqs. 12 and 13. Instead both equations can be replaced by Eq. 15. Thereby the target condition D^T is transferred from the output of the current gate to one of its predecessors and the solver guided toward an early distinction of the faults:

$$D^T \Rightarrow (D_1^T \vee D_2^T \vee \cdots \vee D_m^T) \tag{15}$$

Note that in the shared propagation cone of both faults, D^T is defined as $B^\alpha \oplus B^\beta$. Hence there is no need to encode the G value, reducing the maximal number of encoded values per signal to 3 (see Table 6).

Table 4 Values used for the *backward implication D-chain* in diagnostic test pattern generation

prop. cone f_α	prop. cone f_β	encoded signals
✗	✗	G
✓	✗	G, B^α, D^α
✗	✓	G, B^β, D^β
✓	✓	$G, B^\alpha, B^\beta, D^\alpha, D^\beta$

Table 5 Variation of difference encoding for diagnostic TPG

prop. cone f_α	prop. cone f_β	targeted difference
✓	✗	D^α
✗	✓	D^β
✓	✓	$D^\alpha \oplus D^\beta$

This reduction of encoded values results in a reduction of both search space and formula size compared to the standard backward implication D-chain.

4.4 Indirect D-chain

The indirect D-chain completely refrains from using B values, utilizing instead both the G and D value of a signal. For the context of diagnostic test pattern generation, the used values depend on the propagation cone(s) a signal is contained in, as described in Table 7.

4.5 Hybrid Indirect D-chain

As described in Section 3.5, the hybrid indirect D-chain partly re-calculates the B value, based on an either dynamic or static selection heuristic. The heuristics can be independently calculated and applied for both faults and thereby directly translated to the concept of pattern generation for diagnosis.

5 Evaluation

We evaluated the impact of all previously discussed D-chains on automatic test pattern generation for combinational variants of the largest ITC'99 benchmarks [6], large industrial circuits by NXP as well as artificial cryptographic benchmark circuits based on the advanced encryption standard (AES) [17]. The circuits were synthesized with the 45 nm version of the NanGate cell library [36] which contains a large selection of complex cells. Further information on the benchmarks is listed in Table 8. It should be noted that the AES benchmark circuits are not highly optimized cipher implementations but artificial benchmarks that were generated for the analysis of fault attacks. As such they are purely combinational and without any flip-flops; they are ideal to gage the performance of the presented algorithms on highly complex and deep circuits with many reconvergences.

Table 6 Values used for the *early target backward implication D-chain* in diagnostic test pattern generation

prop. cone f_α	prop. cone f_β	encoded signals
✗	✗	G
✓	✗	G, B^α, D^T
✗	✓	G, B^β, D^T
✓	✓	B^α, B^β, D^T

Table 7 Values used for the *indirect D-chain* in diagnostic test pattern generation

prop. cone f_α	prop. cone f_β	encoded signals
✗	✗	G
✓	✗	G, D^α
✗	✓	G, D^β
✓	✓	G, D^α, D^β

All computations were performed on an Intel Xeon E5-2643 CPU clocked at 3.3 GHz with 64 GB of main memory. The SAT solver *antom* [28] with a timeout of 10 seconds was used as back-end. All methods were incorporated under the *phaeton* framework, introduced in [25].

The ATPG algorithm is used without fault simulation to evaluate the unbiased impact of the different D-chain implementations. Thus, a new Boolean formula is created for every single fault resulting in considerably higher runtimes than those observed in an ATPG with fault simulation. However, only with this strategy a fair comparison of different algorithms is possible, as otherwise the different D-chains would result in different test patterns – as the formulas evaluated by the solver are different – and hence fault simulation could lead to the problem that completely different faults are considered by the different approaches.

Furthermore, for the initial experiments incremental solving is not utilized. While incremental solving does not change the order of computation, it does enforce the propagation of the fault effect to a single specific output. As such it limits the possibilities for the fault propagation and thereby potentially the influence of the D-chains. The influence of incremental solving is analyzed in Section 5.3.

Experimental results for the impact of different D-chain concepts on the total solve time are presented for the stuck-at ATPG, transition-delay ATPG and stuck-at DTPG in Tables 9, 10 and 11 respectively. Changes in total runtime are listed in Table 12 for the stuck-at ATPG.

5.1 Solve Time (ATPG)

For each D-chain type we measured the change in total solve time compared to the basic SAT-based ATPG without any D-chain. The difference between the different ATPG modes lies only with the addition or absence of a D-chain. The experiments were performed for the stuck-at as well as

Table 8 Detailed information about the benchmark circuits used for the evaluation

					Stuck-at ATPG			Transition-delay ATPG		
	Circuit	#Inputs	#Outputs	#Gates	#Fault Instances	#Undetectable	Fault coverage	#Fault Instances	#Undetectable	Fault coverage
ITC'99	b15	485	519	3 395	15 724	199	98.73 %	24 132	2 407	90.03 %
	b17	1 451	1 511	11 345	52 455	642	98.78 %	80 192	10 319	87.13 %
	b18	3 307	3 293	34 936	153 374	75	99.95 %	226 488	36 658	83.81 %
	b20	522	512	5 844	25 202	49	99.81 %	36 444	2 505	93.13 %
	b21	522	512	5 899	25 522	42	99.84 %	37 170	2 437	93.44 %
	b22	735	725	8 144	35 247	68	99.81 %	50 716	3 688	92.73 %
NXP	p35k	2 861	2 229	8 591	43 714	4	99.99 %	63 092	303	99.52 %
	p45k	3 739	2 550	11 413	48 268	5	99.99 %	69 870	1 836	97.37 %
	p78k	3 148	3 484	25 740	129 972	0	100.00 %	171 212	2 080	98.79 %
	p81k	4 029	3 952	44 559	182 916	5	100.00 %	272 158	25 780	90.53 %
	p89k	4 628	4 481	25 209	111 850	120	99.89 %	166 930	11 461	93.13 %
	p100k	5 557	5 489	25 633	115 379	193	99.83 %	166 286	3 888	97.66 %
	p267k	15 426	14 721	47 986	229 405	20	99.99 %	336 164	3 628	98.92 %
	p295k	16 398	16 414	52 366	262 631	1 609	99.39 %	395 860	22 370	94.35 %
	p330k	12 893	12 639	54 287	239 793	695	99.71 %	337 670	3 145	99.07 %
	p378k	15 732	17 420	125 824	653 972	0	100.00 %	851 306	12 404	98.54 %
	p388k	20 449	19 643	118 920	538 848	135	99.97 %	791 748	68 385	91.36 %
AES	2-2-2-8_d	64	32	5 597	23 579	0	100.00 %	35 244	2	99.99 %
	2-2-2-8_e	64	32	4 887	20 916	0	100.00 %	31 362	0	100.00 %
	2-4-4-4_d	128	64	2 056	8 569	0	100.00 %	9 844	0	100.00 %
	2-4-4-4_e	128	64	1 726	7 181	0	100.00 %	8 502	0	100.00 %
	10-2-2-4_d	32	16	1 923	7 947	0	100.00 %	9 886	0	100.00 %
	10-2-2-4_e	32	16	1 936	8 153	0	100.00 %	10 132	0	100.00 %
	10-2-4-4_d	64	32	3 462	14 453	0	100.00 %	17 696	0	100.00 %
	10-2-4-4_e	64	32	3 515	14 974	0	100.00 %	18 286	0	100.00 %

the transition-delay fault model with results summarized in Tables 9 and 10 as well as Figs. 6 and 7.

Generally, most D-chain types provide a large benefit to the overall solve time. The choice of the optimum D-chain depends on both the benchmark class as well as the fault model.

For the ITC'99 and NXP circuits the backward and indirect D-chains usually provide very good results. The cryptographic AES based circuits on the other hand gain the most from an indirect hybrid chain with a dynamic node selection heuristic. The AES circuits have a high combinational depth and only few outputs. The design of the AES cipher furthermore ensures that even a single bit flip can quickly spread through the entire cipher state. This leads to many reconverging paths on which fault effects might cancel each other out. For such circuits, the conventional D-chains appear to be less well suited. The static node selection heuristics gives the fastest results for some of the NXP circuits but is generally slightly outmatched by the other indirect variants.

On average across all circuits, the dynamic hybrid indirect chain also provides the largest gains overall with a speed increase of 71.0% for the stuck-at and 69.5% for the transition-delay fault model.

5.2 Formula Size (ATPG)

Adding information to the original formula increases its size. The respective formula size for each benchmark in stuck-at ATPG is appended in Table 13. Figure 8 shows the average increase across all benchmarks for the different D-chain types.

The backward D-chain is more expensive in terms of additional clauses than the forward D-chain because it utilizes the full equivalence (see Formula (5)) instead of only an implication (Formula (1)) for the D variable.

The size of the indirect encoding is similar to the backward D-chain which is due to the high expenses for gates with many inputs that can have a difference. Utilizing a hybrid encoding slightly reduces the size of the overall

Table 9 Change in total solve time for the different D-chains in the stuck-at ATPG

					Time Difference (%)					
	Circuit	#Fault Instances	Total Memory (MB)	Time (s)	Forward	Backward	Backward + Forward	Indirect	Hybrid Dynamic	Hybrid Static
ITC’99	b15	15 724	95.45	348.46	39.65	−91.84	−90.46	−92.34	**−93.01**	−89.34
	b17	52 455	204.99	847.59	22.80	**−92.27**	−90.15	−91.58	−91.35	−87.56
	b18	153 374	488.17	3 147.57	−6.42	**−87.91**	−85.21	−87.90	−87.52	−83.58
	b20	25 202	116.83	141.13	35.53	−85.53	−80.34	**−86.24**	−82.56	−80.12
	b21	25 522	117.76	162.70	38.46	−85.73	−80.92	**−87.79**	−84.63	−80.44
	b22	35 247	145.93	217.04	31.24	−86.83	−81.39	**−87.93**	−85.00	−79.59
	Average:				26.88	−88.35	−84.75	**−88.97**	−87.34	−83.44
NXP	p35k	43 714	203.66	1 834.64	−41.04	−79.01	−73.81	−78.66	−77.68	**−79.97**
	p45k	48 268	217.98	34.14	1.37	**−67.62**	−64.19	−63.24	−63.75	−61.59
	p78k	129 972	379.65	96.71	−3.47	−69.65	−63.69	**−75.70**	−72.29	−45.96
	p81k	182 916	539.28	542.58	−5.93	−78.31	−75.59	−75.72	**−79.33**	−77.59
	p89k	111 850	401.66	290.37	10.86	**−81.76**	−77.07	−77.85	−78.95	−78.87
	p100k	115 379	422.50	138.56	23.62	**−69.51**	−67.60	−57.34	−63.32	−30.16
	p267k	229 405	888.62	1 243.88	5.35	−89.61	−89.42	**−91.68**	−90.42	−89.60
	p295k	262 631	1 069.65	630.46	15.20	**−77.45**	−74.03	−64.38	−66.20	−66.20
	p330k	239 793	857.55	3 659.29	43.01	−90.22	−89.32	**−90.76**	−89.78	−90.14
	p378k	653 972	1 782.50	1 798.10	15.40	−83.60	−80.27	**−85.90**	−84.64	−47.68
	p388k	538 848	1 753.60	3 768.05	21.23	−79.29	−75.73	−83.40	**−84.66**	−78.95
	Average:				7.78	**−78.73**	−75.52	−76.78	−77.36	−67.88
AES	2-2-2-8_d	23 579	109.98	429.20	46.44	14.34	123.95	−38.15	**−47.36**	8.96
	2-2-2-8_e	20 916	100.20	357.05	−1.10	−58.46	−42.39	**−77.92**	−76.81	−18.99
	2-4-4-4_d	8 569	61.39	30.47	102.57	75.24	337.07	27.71	**−25.62**	3.91
	2-4-4-4_e	7 181	58.60	19.33	−2.13	−65.47	−61.29	**−84.90**	−81.96	−55.22
	10-2-2-4_d	7 947	60.75	119.84	46.51	−9.25	38.93	17.05	**−46.47**	36.83
	10-2-2-4_e	8 153	60.39	85.65	40.17	−16.46	3.84	−8.36	**−35.73**	13.77
	10-2-4-4_d	14 453	76.89	712.56	50.99	−4.98	98.32	29.61	**−43.39**	35.83
	10-2-4-4_e	14 974	73.09	619.73	46.36	−16.47	36.41	−20.62	**−42.88**	20.73
	Average:				41.23	−10.19	66.85	−19.45	**−50.03**	5.73

Maximal solve time reductions are emphasized

formula because some of these expensive gates are handled in the cheaper, conventional manner.

The relative impact of adding a D-chain turns out to be lower for the transition-delay fault model, probably since the formula also contains additional information for the first time frame.

5.3 Incremental Solving

Utilizing the SAT solver in an incremental manner greatly increases the overall speed of the ATPG algorithm. Unlike the normal SAT-based ATPG discussed in Section 2, the fault propagation is much stronger enforced, since the fault has to be visible at one particular output out of the modeled outputs. Hence, the gain of adding a D-chain as, yet another, fault propagation support mechanism is lower. Nonetheless, both the total computation time as well as the total solve time are generally improved by adding D-chains.

The total solve time for the stuck-at ATPG is improved by 35% on average for the ITC’99 and NXP circuits with both the backward implication or indirect D-chain (without incremental solving the total solve time is improved by about 89% and 79% for these circuit classes, respectively). For the AES benchmarks, the largest gain of 25% on average was, again, achieved with the hybrid indirect chain with the dynamic selection heuristic (50% without incremental solving).

For the transition-delay ATPG the results are similar, with an average decrease of around 46% for the ITC’99 and NXP circuits (86% and 74% without incremental solving, respectively) and 25% for the AES circuits (51% without incremental solving).

Table 10 Change in total solve time for the different D-chains in the transition-delay ATPG

					Time Difference (%)					
	Circuit	#Fault Instances	Total Memory (MB)	Time (s)	Forward	Backward	Backward + Forward	Indirect	Hybrid Dynamic	Hybrid Static
ITC'99	b15	24 132	98.88	1 179.73	29.48	−91.77	−90.89	**−92.58**	−92.37	−89.67
	b17	80 192	207.64	2 704.74	14.60	−89.19	−88.07	**−89.50**	−89.44	−86.44
	b18	226 488	521.85	9 848.01	−11.69	−85.73	−83.67	**−86.78**	−86.28	−82.23
	b20	36 444	133.67	572.17	20.73	−83.63	−80.94	**−84.56**	−83.47	−80.63
	b21	37 170	134.57	696.17	15.89	−83.53	−81.11	**−85.15**	−83.51	−80.22
	b22	50 716	157.46	884.72	19.19	−80.29	−78.26	**−81.21**	−80.51	−76.66
	Average:				14.70	−85.69	−83.82	**−86.63**	−85.93	−82.64
NXP	p35k	63 092	213.20	16 097.94	−35.01	−77.40	−76.08	−74.86	−76.67	**−77.51**
	p45k	69 870	221.97	123.54	4.77	**−64.95**	−60.71	−61.79	−64.43	−57.22
	p78k	171 212	382.83	300.30	45.16	−70.41	−68.69	**−70.93**	−67.16	−50.25
	p81k	272 158	553.68	4 779.02	9.10	**−74.80**	−73.47	−71.79	−72.85	−73.07
	p89k	166 930	412.86	1 276.92	2.72	−69.88	−69.50	−66.92	−70.00	**−70.73**
	p100k	166 286	434.80	520.99	6.64	**−63.57**	−62.53	−51.66	−61.01	−41.32
	p267k	336 164	905.25	7 867.02	−8.27	−80.77	−81.55	**−84.25**	−83.99	−83.76
	p295k	395 860	1 099.52	3 513.08	−1.27	**−68.66**	−66.67	−65.81	−66.69	−66.25
	p330k	337 670	887.60	28 862.20	−8.66	**−90.06**	−89.95	−89.72	−89.89	−89.91
	p378k	851 306	1 845.67	7 600.83	37.43	−89.36	−88.31	**−89.61**	−89.24	−67.32
	p388k	791 748	1 785.75	32 292.94	−2.61	−66.69	−63.44	−65.64	**−67.60**	−64.31
	Average:				4.55	**−74.23**	−72.81	−72.09	−73.59	−67.42
AES	2-2-2-8_d	35 244	110.23	762.86	43.63	7.49	107.75	−43.05	**−49.92**	7.64
	2-2-2-8_e	31 362	104.54	635.88	−4.49	−63.89	−49.45	**−80.15**	−78.88	−24.89
	2-4-4-4_d	9 844	61.31	43.04	99.72	48.42	296.25	18.42	**−37.73**	−0.63
	2-4-4-4_e	8 502	58.54	29.20	−4.02	−70.10	−65.28	**−86.46**	−85.13	−57.89
	10-2-2-4_d	9 886	63.65	220.80	33.30	−17.54	18.19	0.85	**−44.30**	15.66
	10-2-2-4_e	10 132	60.37	152.21	34.04	−18.95	−5.32	−14.25	**−35.50**	14.37
	10-2-4-4_d	17 696	74.18	1 470.09	36.00	−14.35	62.17	6.74	**−38.49**	18.70
	10-2-4-4_e	18 286	75.82	1 242.51	31.54	−27.58	9.32	−29.34	**−42.17**	12.68
	Average:				33.72	−19.56	46.70	−28.40	**−51.51**	−1.80

Maximal solve time reductions are emphasized

These results clearly show that even with more advanced modeling and solving techniques, D-chains still provide a substantial increase in solving speed.

5.4 DTPG Results

Similar to the ATPG algorithm, DTPG is evaluated without fault simulation to get the unbiased impact of the different D-chain implementations, as a coincidental number of fault pairs could accidentally be distinguished with simulation of a calculated test pattern.

Thus, a new Boolean formula is created for every single fault pair (except for already merged faults), which is not feasible. To give a comprehensive analysis on large benchmark sets, we therefore evaluated the different D-chain concepts for DTPG without simulation for a fixed set of 100,000 random fault pairs for each benchmark.

Similar to the ATPG solve time evaluation, we measured the change in total solve time compared to the basic SAT-based DTPG without the use of any D-chain. For the experiments we focus on the stuck-at fault model, as we already presented a comparison between different fault models for ATPG.

Figure 9 shows the total solve time results of the diagnostic test pattern generation. It can be seen that except for the forward implication D-chain every analyzed D-chain shows a positive impact on the total solve time for the ITC'99 and NXP benchmark set, reducing the total solve time on average by at least 67%. For the ITC'99 benchmarks alone, the results are even better with an average reduction of the total

Table 11 Change in total solve time for the different D-chains in the stuck-at DTPG

					Time Difference (%)					
	Circuit	#Fault Instances	Total Memory (MB)	Time (s)	Forward	Backward	Early Target Backward	Indirect	Hybrid Dynamic	Hybrid Static
ITC'99	b15	15 724	78.04	3 201.62	33.94	−90.43	−90.91	**−92.21**	−92.01	−88.83
	b17	52 455	152.58	2 715.00	26.45	−91.35	**−91.49**	−90.91	−91.37	−89.29
	b18	153 374	323.27	3 060.34	5.13	−89.19	**−89.58**	−88.71	−89.15	−86.58
	b20	25 202	94.50	1 342.66	31.64	−86.54	−87.12	**−88.05**	−86.60	−85.43
	b21	25 522	96.61	1 521.35	25.86	−87.74	−88.03	**−89.50**	−87.69	−85.55
	b22	35 247	114.85	1 515.95	15.33	−88.28	−88.61	**−89.56**	−88.35	−86.01
	Average:				23.06	−88.92	−89.29	**−89.82**	−89.19	−86.95
NXP	p35k	43 714	146.38	2 272.19	5.21	**−56.93**	−53.54	−54.87	−52.52	−53.26
	p45k	48 268	157.05	127.47	12.37	−68.14	**−70.73**	−65.15	−68.62	−64.05
	p78k	129 972	717.34	160.09	−10.00	−65.25	−66.21	**−69.10**	−65.70	−40.88
	p81k	182 916	557.13	497.95	15.20	−76.51	−77.38	−78.03	**−78.19**	−77.27
	p89k	111 850	1 121.20	384.15	14.17	−81.00	**−81.76**	−80.97	−81.61	−81.47
	p100k	115 379	284.52	162.49	30.47	−66.46	**−68.57**	−51.60	−66.02	−49.83
	p267k	229 405	574.03	731.08	19.09	−88.32	−88.84	**−89.99**	−89.60	−88.85
	p295k	262 631	717.34	543.69	23.90	−83.06	**−83.81**	−79.74	−80.61	−80.93
	p330k	239 793	557.13	2 035.46	20.42	−89.04	−89.91	**−90.54**	−89.80	−89.60
	p378k	653 972	1 121.20	194.41	−6.78	−56.68	−60.56	**−65.41**	−65.40	−35.58
	p388k	538 848	1 178.13	816.78	20.16	−74.14	−73.62	−80.43	**−81.25**	−78.37
	Average:				13.11	−73.23	−74.08	−73.26	**−74.48**	−67.28
AES	2-2-2-8_d	23 579	101.41	2 905.68	47.85	26.88	4.40	−33.92	**−45.75**	43.48
	2-2-2-8_e	20 916	104.06	2 500.78	17.29	−54.78	−55.63	**−75.91**	−75.03	−8.61
	2-4-4-4_d	8 569	56.32	1 059.12	57.41	23.47	22.98	−19.63	**−61.60**	4.73
	2-4-4-4_e	7 181	53.87	512.18	18.12	−53.66	−63.56	**−86.49**	−84.64	−48.90
	10-2-2-4_d	7 947	59.39	1 619.37	25.43	63.75	40.61	59.73	**20.24**	56.65
	10-2-2-4_e	8 153	61.47	978.10	49.55	61.92	45.89	35.08	**30.51**	54.94
	10-2-4-4_d	14 453	66.37	6 032.42	44.63	104.87	60.09	63.49	**40.95**	72.43
	10-2-4-4_e	14 974	65.98	4 846.07	56.16	50.80	29.17	22.81	**21.93**	62.26
	Average:				39.56	27.90	10.49	−4.36	**−19.17**	29.62

Maximal solve time reductions are emphasized

solve time by around 87% to 89%. The hybrid static D-chain shows the second least solve time improvement on the industrial benchmark set (-67%, NXP), only outperforming the forward implication D-chain, where the other 4 D-chains provide an average solve time reduction of 73% to 75%. The AES benchmarks present a strongly different result, where only the indirect and the hybrid dynamic D-chain encodings reduce the total solve time on average.

As additional information is added to the original formula, the size increases. The conventional backward implication D-chain both has a comparably high number of different signal values (see Table 4) and utilizes the full equivalence (see Formula (11)) for each D-value, thus, as Fig. 10 shows, needs the most additional clauses of all presented DTPG D-chains for each benchmark set. In comparison to the other D-chains, the hybrid encodings and the forward implication D-chain offer an overall small increase in formula size.

5.5 Total Runtime

The previous results focused on the solve time which does not include the generation of a Boolean formula for every fault, transmitting this formula to the SAT solver, and finally extracting the test pattern.

When taking this overhead into account, the influence of the different D-chain versions decreases because the formula generation often requires more time than the actual solving, especially for the easy-to-solve stuck-at problems.

Nonetheless, D-chains still drastically increase the solving speed, with an average gain of 32.6% on the stuck-at and 42.3% on the transition-delay ATPG for the dynamic hybrid indirect chain and furthermore 35.9% on DTPG for the early target backward D-chain. The total runtimes for stuck-at ATPG are shown in Table 12, the total runtimes for transition-delay ATPG and DTPG are omitted for brevity.

Table 12 Change in total runtime for the different D-chains in the stuck-at ATPG

					Time Difference (%)					
	Circuit	#Fault Instances	Total Memory (MB)	Total Runtime (s)	Forward	Backward	Backward + Forward	Indirect	Hybrid Dynamic	Hybrid Static
ITC’99	b15	15 724	95.45	392.86	37.66	−77.59	−75.29	−76.63	**−78.46**	−75.16
	b17	52 455	204.99	1 023.54	19.02	**−73.53**	−69.25	−71.99	−69.99	−68.56
	b18	153 374	488.17	3 959.88	−6.44	**−68.85**	−64.92	−68.33	−67.97	−65.69
	b20	25 202	116.83	196.72	32.35	**−55.22**	−44.54	−54.70	−48.52	−50.15
	b21	25 522	117.76	221.37	34.36	**−59.49**	−50.96	−58.36	−54.81	−52.82
	b22	35 247	145.93	297.34	28.61	**−60.39**	−48.26	−58.01	−54.50	−51.77
	Average:				24.26	**−65.85**	−58.87	−64.67	−62.37	−60.69
NXP	p35k	43 714	203.66	2 496.78	−39.01	−54.58	−46.82	−51.93	−50.94	**−55.13**
	p45k	48 268	217.98	122.93	−6.50	−2.34	−2.06	−1.97	4.53	**−7.46**
	p78k	129 972	379.65	453.45	**−10.64**	−1.84	8.20	−2.46	−6.26	5.01
	p81k	182 916	539.28	1 429.63	−7.65	−15.31	−10.65	−12.05	**−21.35**	−17.33
	p89k	111 850	401.66	668.33	5.21	**−19.63**	−9.58	−12.72	−12.17	−11.03
	p100k	115 379	422.50	490.88	8.42	8.96	8.67	**5.04**	15.68	12.74
	p267k	229 405	888.62	2 695.66	3.65	−24.31	−26.96	−29.25	−24.52	**−29.63**
	p295k	262 631	1 069.65	2 659.09	2.41	−0.44	−0.40	4.62	2.63	**−0.69**
	p330k	239 793	857.55	5 845.61	27.23	−47.89	−47.29	**−49.11**	−46.54	−48.50
	p378k	653 972	1 782.50	8 163.10	2.93	**−4.68**	−4.67	−4.10	−3.27	1.44
	p388k	538 848	1 753.60	9 906.86	5.39	−22.01	−20.19	−23.08	−22.35	**−23.78**
	Average:				−0.78	**−16.73**	−13.79	−16.09	−14.96	−15.85
AES	2-2-2-8_d	23 579	109.98	604.20	38.01	19.04	102.23	−9.64	**−24.38**	16.07
	2-2-2-8_e	20 916	100.20	510.48	2.34	−36.16	−23.22	−48.04	**−49.45**	−8.82
	2-4-4-4_d	8 569	61.39	42.75	79.70	60.96	252.53	25.76	**−11.89**	8.59
	2-4-4-4_e	7 181	58.60	27.79	0.48	−43.16	−38.56	**−57.29**	−56.18	−37.47
	10-2-2-4_d	7 947	60.75	139.63	43.84	−3.49	41.14	20.30	**−34.89**	36.32
	10-2-2-4_e	8 153	60.39	103.35	36.50	−10.02	9.57	−1.12	**−25.77**	15.38
	10-2-4-4_d	14 453	76.89	778.75	48.86	−1.98	94.25	30.23	**−37.00**	35.36
	10-2-4-4_e	14 974	73.09	679.66	43.96	−13.07	36.50	−16.01	**−37.19**	20.89
	Average:				36.71	−3.49	59.30	−6.98	**−34.60**	10.79

Maximal runtime reductions are emphasized

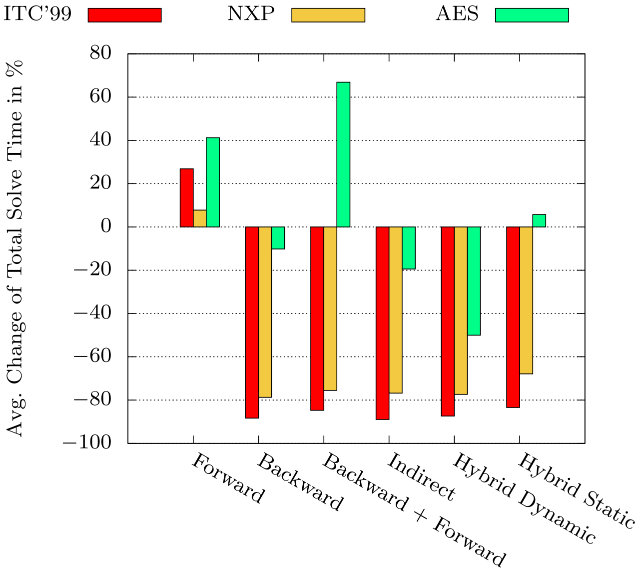

Fig. 6 The change in total solve time for the different D-chains and circuit groups in the *stuck-at* fault model (ATPG)

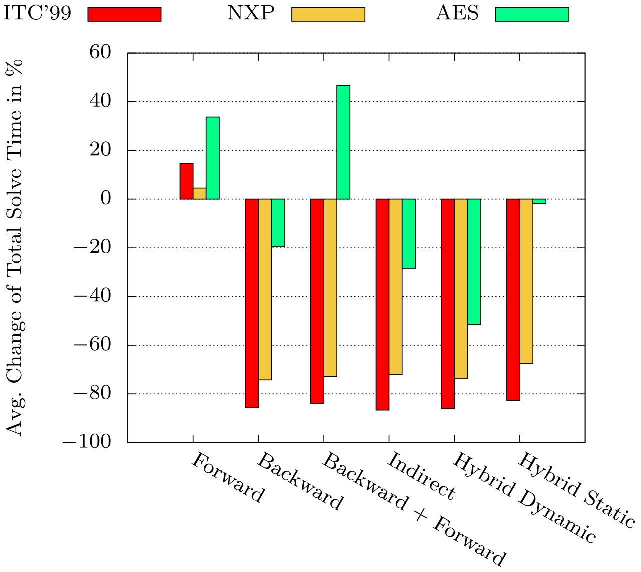

Fig. 7 The change in total solve time for the different D-chains and circuit groups in the *transition-delay* fault model (ATPG)

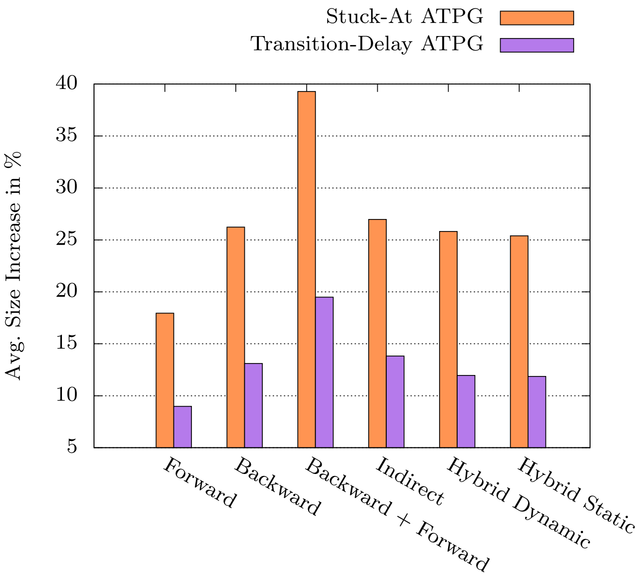

Fig. 8 Increase in formula size for the different D-chains and fault models (ATPG)

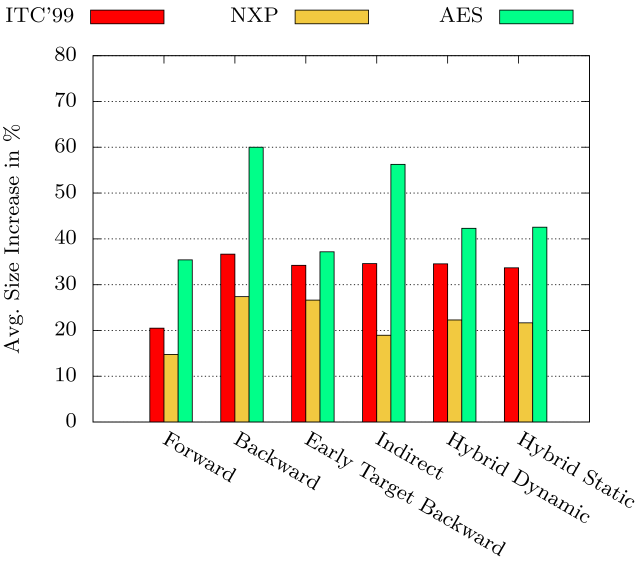

Fig. 10 Increase in formula size for the different D-chains (DTPG)

5.6 Results Summary

The experimental results in Tables 9 and 10 clearly show that D-chains significantly reduce the total solve time as well as the total computation time in SAT-based ATPG both in the conventional mode and when utilizing the solver incrementally. Our newly introduced indirect D-chain and its variants often provide similar or even better results than the previously known D-chains and achieve the highest speedup on average.

However, the gain of the different D-chains depends on the circuit type. For the analyzed ITC'99 and NXP circuits the backward and indirect D-chains generally give the best results for ATPG, whereas for the cryptographic AES based circuits only the hybrid indirect D-chain with the dynamic selection heuristic results in large decreases in computation time. Furthermore, the analysis of the ATPG results shows that fault model also affects the D-chain gains. For the more difficult transition-delay ATPG the gain through D-chains is on average about 9.6%. larger than for the stuck-at ATPG.

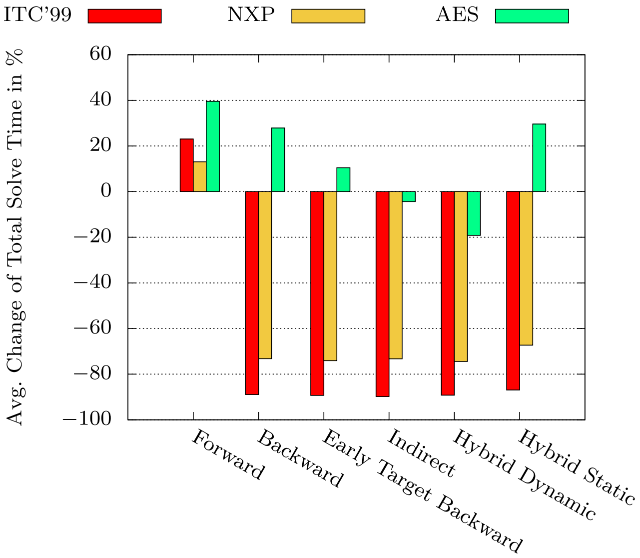

Fig. 9 The change in total solve time for the different D-chains and circuit groups (DTPG)

Table 11 shows the experimental results for the diagnostic test pattern generation.

Here, the influence of the circuit type is even more apparent than it was in the ATPG analysis. For both ITC'99 and NXP circuits every analyzed D-chain reduces the total solve time for every single benchmark. In contrast to these results, diagnosis for the cryptographic AES circuits benefits solely from the indirect and the hybrid dynamic D-chains whereas the remaining variants are actually slowing down the solver.

In addition to the underlying circuit type, the choice of which D-chain to implement depends strongly on the application. For both ATPG and DTPG we presented easy-to-implement D-chains with a good overall speedup, as well as complex D-chain variants, where the hybrid dynamic D-chain shows the best overall solve time reduction. Additionally D-chains were presented, that offer a trade-off between straightforward implementation and great overall speedup, e.g. the indirect D-chain.

The gain of the early target backward D-chain in comparison to the classic backward D-chain is surprisingly small, considering how many fewer variables were used in the encoding. Nonetheless, it still outperforms the backward D-chain on almost any circuit and provides an easy-to-implement alternative to the indirect D-chain variants that

Table 13 Change in the number of clauses per fault in the stuck-at ATPG

		#Clauses			#Clauses Difference					
					Forward	Backward	Backward + Forward	Indirect	Hybrid Dynamic	Hybrid Static
	Circuit	min	avg	max	avg	avg	avg	avg	avg	avg
ITC'99	b15	19	8 743.81	37 692	**1 471.01**	2 210.81	**3 329.49**	2 683.18	2 619.34	2 575.24
	b17	19	7 458.15	44 577	**1 252.18**	1 897.73	**2 835.07**	2 325.42	2 307.05	2 269.66
	b18	19	8 621.55	61 145	**1 291.73**	1 911.13	**2 863.29**	2 082.65	2 267.30	2 236.53
	b20	11	6 061.82	42 804	**1 217.74**	1 793.58	**2 670.90**	1 990.78	2 096.26	2 024.11
	b21	11	6 325.98	44 897	**1 253.09**	1 832.89	**2 734.52**	2 079.18	2 146.73	2 082.47
	b22	14	5 500.05	57 669	**1 037.80**	1 515.43	**2 272.45**	1 604.19	1 726.85	1 670.85
NXP	p35k	21	21 949.27	49 384	**1 736.73**	2 548.65	**3 776.15**	2 683.59	3 122.25	3 068.50
	p45k	7	1 562.48	86 113	**169.93**	253.29	**383.18**	234.70	263.62	256.11
	p78k	7	1 332.53	13 419	278.48	440.93	**654.17**	**246.00**	332.76	323.19
	p81k	16	3 084.35	229 180	**410.01**	546.58	**850.74**	519.89	677.17	645.61
	p89k	11	2 101.25	63 677	**307.21**	472.63	**709.71**	492.59	559.10	542.66
	p100k	7	1 732.85	88 349	**275.44**	391.29	**591.97**	394.00	413.67	404.56
	p267k	7	2 151.39	112 524	**316.04**	448.46	**712.35**	386.01	465.23	446.46
	p295k	10	2 748.49	635 146	**257.75**	392.31	**592.72**	459.38	483.06	474.72
	p330k	7	5 671.08	166 710	**765.51**	1 030.60	**1 650.16**	906.80	1 131.35	1 112.85
	p378k	7	1 343.73	887 384	282.79	440.00	**656.47**	**243.04**	330.17	320.36
	p388k	16	3 114.21	829 845	**459.80**	657.62	**1 000.98**	652.35	775.84	745.30
AES	2-2-2-8_d	13 722	26 572.04	50 323	**7 430.36**	11 071.23	16 063.42	**21 376.40**	11 417.71	11 380.97
	2-2-2-8_e	14 461	24 407.16	39 924	**2 412.80**	3 798.63	5 453.43	**6 201.28**	3 891.84	3 899.27
	2-4-4-4_d	2 697	5 884.01	13 895	1 483.70	2 192.25	**3 241.86**	**1 357.99**	1 521.26	1 511.26
	2-4-4-4_e	2 437	4 647.26	8 523	342.01	628.10	**918.38**	**0.43**	68.28	66.72
	10-2-2-4_d	6 425	10 311.83	14 076	**3 529.69**	4 844.21	**7 231.55**	4 912.37	4 259.65	4 252.92
	10-2-2-4_e	6 240	9 122.88	13 334	**2 229.62**	3 121.34	**4 641.84**	3 537.33	2 615.93	2 688.07
	10-2-4-4_d	10 318	17 591.45	25 194	**5 721.80**	7 875.93	**11 752.51**	7 812.29	6 825.15	6 817.57
	10-2-4-4_e	10 241	15 713.68	23 536	**3 584.45**	5 013.15	**7 465.57**	5 515.02	4 107.18	4 225.80

Maximal and minimal differences in number of clauses are emphasized

require a more complex gate modeling. Furthermore in our experiments the early target backward D-chain showed the greatest reduction in total DTPG runtime, which can be attributed to its lightweight structure; therefore it can be quickly constructed.

6 Conclusion and Future Work

We analyzed the effect of different D-chain implementations on SAT-based ATPG for the widely used stuck-at and transition-delay fault models and diagnostic TPG for the stuck-at fault model.

In addition to already established D-chains we introduced and analyzed different types of indirect D-chain implementations, which avoid overhead by removing redundant information and only focusing on the difference between the fault-free and faulty circuit.

We also introduced the early target backward implication D-chain specifically for DTPG, which dynamically defines a difference value depending on the topological position of the corresponding signal line, to actively guide the solver toward a fault-distinguishing test pattern.

Experimental results demonstrate the significant benefits of the newly introduced D-chain concepts for ATPG and DTPG on different benchmark sets, with an average solve time reduction of 70% for ATPG and 54% for DTPG.

In the future we plan on extending the idea of indirect D-chains both beyond simple Boolean logic and toward more complex fault models and to analyze the gains of the different D-chains in these areas. Furthermore, we want to evaluate the benchmark class of cryptographic circuits in more detail and investigate the impact of selecting solvers especially tuned for cryptographic circuits e.g. CryptoMiniSat [30] on the effectiveness of D-chains.

Overall, this article clearly shows that D-chains are a vital part of a fast and efficient SAT-based ATPG and DTPG flow and are well worth the extra effort during the formula generation.

References

1. Boros E, Hammer PL (2002) Pseudo-Boolean optimization. Discret Appl Math 123(1-3):155. https://doi.org/10.1016/S0166-218X(01)00341-9
2. Burchard J, Neubauer F, Raiola P, Erb D, Becker B (2017) Evaluating the effectiveness of D-chains in SAT-based ATPG. In: Proceedings 18th IEEE Latin American Test Symposium, pp 1–6
3. Cheng KT (1993) Transition fault testing for sequential circuits. IEEE Trans Comput Aided Des Integr Circuits Syst 12(12):1971
4. Chen H, Marques-Silva J (2009) TG-PRO: a new model for SAT-based ATPG. In: Proceedings of the IEEE International High Level Design Validation and Test Workshop, pp 76–81
5. Chen H, Marques-Silva J (2013) A two-variable model for SAT-based ATPG. IEEE Trans Comput Aided Des Integr Circuits Syst 32(12):1943
6. Corno F, Reorda MS, Squillero G (2000) RT-level ITC'99 benchmarks and first ATPG results. IEEE Des Test 17(3):44. https://doi.org/10.1109/54.867894
7. Drechsler R, Eggergluss S, Fey G, Glowatz A, Hapke F, Schloeffel J, Tille D (2008) On acceleration of SAT-based ATPG for industrial designs. IEEE Trans Comput Aided Des Integr Circuits Syst 27(7):1329
8. Eén N, Sörensson N (2004) An extensible SAT-solver. In: Proceedings Theory and Applications of Satisfiability Testing (SAT)
9. Eggersglüß S, Drechsler R (2011) As-Robust-As-Possible test generation in the presence of small delay defects using pseudo-Boolean optimization. In: Proceedings of the Design, Automation Test in Europe, pp 1–6
10. Eggersglüß S, Krenz-Bååth R, Glowatz A, Hapke F, Drechsler R (2012) A new SAT-based ATPG for generating highly compacted test sets. In: IEEE DDECS, pp 230–235
11. Erb D, Scheibler K, Sauer M, Becker B (2014) Efficient SMT-based ATPG for interconnect open defects. In: Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE), pp 1–6
12. Erb D, Scheibler K, Kochte MA, Sauer M, Wunderlich HJ, Becker B (2014) Test pattern generation in presence of unknown values based on restricted symbolic logic. In: Proceedings of the International Test Conference
13. Erb D, Scheibler K, Kochte MA, Sauer M, Wunderlich HJ, Becker B (2016) Mixed 01X-RSL-Encoding for fast and accurate ATPG with unknowns. In: Proceedings of the Asia and South Pacific Design Automation Conference, pp 749–754
14. Fey G, Shi J, Drechsler R (2006) Efficiency of multi-valued encoding in SAT-based ATPG. In: Proceedings of the 36th International Symposium on Multiple-Valued Logic (ISMVL'06), pp 25–25
15. Fujiwara H, Shimono T (1983) On the acceleration of test generation algorithms. IEEE Trans Comput c-32(12):1137–1144
16. Galey JM, Norby RE, Roth JP (1961) Techniques for the diagnosis of switching circuit failures. In: Proceedings of the 2nd Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1961), pp 152–160
17. Gay M, Burchard J, Horacek J, Ekossono ASM, Schubert T, Becker B, Polian I, Kreuzer M, Small scale AES (2016) Toolbox: algebraic and propositional formulas, circuit-implementations and fault equations. In: Proceedings of the FCTRU
18. Goel P (1981) An implicit enumeration algorithm to generate tests for combinational logic circuits. IEEE Trans Comput C-30(3): 215
19. Hsieh ER, Rasmussen RA, Vidunas LJ, Davis WT Delay test generation. In: Proceedings of the 14th Design Automation Conference (IEEE Press, 1977), DAC '77, pp 486–491
20. Larrabee T (1992) Test pattern generation using Boolean satisfiability. IEEE Trans Comput Aided Des 11(1):4–15
21. Raiola P, Erb D, Reddy SM, Becker B (2017) Accurate diagnosis of interconnect open defects based on the robust enhanced aggressor victim model. In: Proceedings of the 30th International Conference on VLSI Design and 16th International Conference on Embedded Systems, pp 135–140
22. Roth JP (1966) Diagnosis of automata failures: a calculus and a method. IBM J Res Dev 10(4):278
23. Rudnick EM, Fuchs WK, Patel JH (1992) Diagnostic fault simulation of sequential circuits. In: Proceedings IEEE International Test Conference, pp 178–186
24. Sauer M, Czutro A, Polian I, Becker B (2012) Small-delay-fault ATPG with waveform accuracy. In: Proceedings of the Int'l Conference on CAD, pp 30–36
25. Sauer M, Becker B, Polian I (2016) PHAETON: a SAT-based framework for timing-aware path sensitization. IEEE Trans Comput 65(6):1869
26. Savir J, Patil S (1994) Broadside delay test. IEEE Trans Comput Aided Des Integr Circuits Syst 13(8):1057
27. Scheibler K, Erb D, Becker B (2016) Accurate CEGAR-based ATPG in presence of unknown values for large industrial designs. In: Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE)
28. Schubert T, Reimer S (2016) antom, in https://projects.informatik.uni-freiburg.de/projects/antom
29. Shi J, Fey G, Drechsler R, Glowatz A, Schloffel J, Hapke F (2005) Experimental studies on SAT-based test pattern generation for industrial circuits. In: Proceedings of the 6th International Conference on ASIC, vol 2
30. Soos M, Nohl K, Castelluccia C (2009) Extending SAT solvers to cryptographic problems. In: Proceedings of the Theory and Applications of Satisfiability Testing, pp 244–257
31. Stephan P, Brayton RK, Sangiovanni-Vincentelli AL (1996) Combinational test generation using satisfiability. IEEE Trans Comput Aided Des Integr Circ Syst 15(9):1167–1176
32. Tafertshofer P, Ganz A (1999) SAT based ATPG using fast justification and propagation in the implication graph. In: Proceedings IEEE/ACM International Conference on Computer-Aided Design, pp 139–146
33. Tille D, Drechsler R (2008) Incremental SAT Instance generation for SAT-based ATPG. In: Proceedings of the 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, pp 1–6
34. Tille D, Eggersglüß S, Drechsler R (2010) Incremental solving techniques for SAT-based ATPG. IEEE Trans Comput Aided Des Integr Circ Syst 29(7):1125
35. Tseitin G (1968) On the complexity of derivation in propositional calculus studies in constructive mathematics and mathematical logic
36. Si2. NanGate FreePDK45 generic open cell library, v1.3. http://www.si2.org/openeda.si2.org/projects/nangatelib
37. Yang K, Cheng KT, Wang LC (2004) Trangen: a SAT-based ATPG for path-oriented transition faults. In: Proceedings of the ASP-DAC: Asia and South Pacific Design Automation Conference, pp 92–97

Pascal Raiola received the B.Sc. degree in mathematics and the M.Sc. degree in computer science from the University of Freiburg, Breisgau, Germany, in 2012 and 2015, respectively. Since 2016 he has been with the group of Computer Architecture of Prof. Bernd Becker. His research interests include hardware security, test and diagnosis under multi-valued logic, SAT applications and data dependence.

Jan Burchard received his bachelor and master degree in computer science from the University of Freiburg, Breisgau, Germany, in 2013 and 2015, respectively. Since 2015 he is pursuing his Ph.D. under the supervision of Prof. Bernd Becker. His research interests include automatic test pattern generation for advanced fault models and SAT-solving techniques as well as hardware security.

Felix Neubauer received the B.Sc. degree and the M.Sc. degree in computer science from the University of Freiburg, Breisgau, Germany, in 2012 and 2015, respectively. Since 2015 he has been with the group of Computer Architecture of Prof. Bernd Becker. His research interests include SAT/SMT and its applications in verification and test.

Dominik Erb received the master's and Ph.D. degree in computer science from the University of Freiburg, Breisgau, Germany, in 2013 and 2016, respectively. His research interests include automatic test pattern generation in presence of unknown values as well as interconnect open defects, defect-based testing, and fault-diagnosis. He is currently working in the Project Management at Infineon Technologies, Neubiberg, Bavaria, Germany.

Bernd Becker is a full professor at the Faculty of Engineering, University of Freiburg, Germany. His research activities include design, test and verification methods for embedded systems and nanoelectronic circuitry. He is a co-speaker in the DFG Transregional Research Center “Automatic Analysis and Verification of Complex Systems” and a director in the Centre for Security and Society, Freiburg. He is a fellow of the IEEE and a member of Academia Europaea.