An effective Approach to Automatic Functional Processor Test Generation for Small-Delay Faults

Andreas Riefert * Lyl Ciganda † Matthias Sauer * Paolo Bernardi † Matteo Sonza Reorda † Bernd Becker *

* Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 051
79110 Freiburg, Germany
{riefert|sauerm|becker}@informatik.uni-freiburg.de

† Politecnico di Torino
Corso Duca degli Abruzzi 24
10129 Torino, Italy
{lyl.ciganda|paolo.bernardi|matteo.sonzareorda}@polito.it

***Abstract*— Functional microprocessor test methods provide several advantages compared to DFT approaches, like reduced chip cost and at-speed execution. However, the automatic generation of functional test patterns is an open issue.**

In this work we present an approach for the automatic generation of functional microprocessor test sequences for small-delay faults based on Bounded Model Checking. We utilize an ATPG framework for small-delay faults in sequential, non-scan circuits and propose a method for constraining the input space for generating functional test sequences (i.e., test programs). We verify our approach by evaluating the miniMIPS microprocessor. In our experiments we were able to reach over 97 % fault efficiency.

To the best of our knowledge, this is the first fully automated approach to functional microprocessor test for small-delay faults.

I. Introduction

Test of processor devices and cores is a major challenge, whose economical importance is growing due to the increasing role that processor-based systems play in many applications. Technical requirements, economical constraints, regulations and standards (especially when the system is used in safety-critical applications) force companies to develop solutions allowing to effectively identify faulty components before they are deployed in the field.

In the past years, the processor test challenge was mostly faced resorting to Design for Testability (DfT) solutions, including scan and BIST. However, there are scenarios where these solutions need to be substituted by a functional approach, which only acts on functional inputs and only monitors functional outputs, without any usage of DfT structures. In particular, the functional approach may be used in those cases when at-speed testing is crucial, or when DfT cannot be used (e.g., because the existing core does not support it and cannot be modified), or when the test must be performed during the operational phase. Moreover, some researchers have shown that different faults are detected by these two strategies [1]. When the Circuit Under Test (or CUT) is a processor, the functional approach requires a proper test program to be executed and the behavior of the processor to be observed (e.g., by looking at the processor outputs, or at the results produced by the program in memory). This approach is often referred to as Software-Based Self-Test (or SBST) [2].

978-3-9815370-2-4/DATE14/©2014 EDAA

A major issue when applying SBST lies in the method to generate test programs. Although the first guidelines to manually perform this task go back to several decades ago [3], techniques to automate are still not developed enough and several research efforts are on-going, targeting both stuck-at [4] and delay faults [5]. In [6] a technique was recently introduced to generate test patterns for small-delay faults, whose importance to achieve high defect coverage has been largely recognized in industry [7]. The method was developed resorting to Bounded Model Checking (BMC) and its effectiveness was validated on the ITC 99 benchmark circuits [8].

When targeting functional test, traditional ATPG techniques must be adapted by adding new constraints: the test is performed by forcing the processor to execute legal instructions that must preliminary be fetched from memory complying with the bus protocol, and the usage of some input signals (e.g., the reset and the interrupt ones) must be suitably limited. Hence, the ATPG must generate a sequence of input values allowing the processor to fetch and execute legal instructions, and faults are detected when they reach the functional output signals. Moreover, a number of faults that can be tested using DfT techniques turn into untestable; this phenomenon becomes even more relevant when the functional test is performed during the operational phase, since even stricter constraints exist in that case [9].

In this work we extend the approach of [6] to the case of functionally tested processors and propose a method to effectively generate functional test programs. This means introducing constraints on the possible inputs used during test: only functional inputs are used and functional outputs are observed and the test is performed by forcing the processor to execute a proper test program.

The contribution of this paper lies in the implementation of a functional ATPG for processors targeting small-delay faults. To the best of our knowledge, this is the first method able to succesfully automate the generation of test programs for small-delay faults for a whole processor. We provide the infrastructure to utilize a *Validity Checker Module (VCM)*, similar to the concept of *Virtual Constraint Circuits (VCC)* [10]. The VCM allows a user to easily specify requirements for the generated test pattern sequences, which enables the restriction of the patterns to functional inputs. Furthermore, it enables the proof of untestability under the specified requirements. Finally we implemented a guided

search in the ATPG process, which enables a high fault efficiency.

We demonstrate the applicability of our approach by evaluating the complete miniMIPS processor [11]. A VCM with several requirements was specified in VHDL and applied. In our experiments we were able to reach a fault efficiency of over 97 % with reasonable CPU time requirements.

The remainder of the paper is organized as follows. Section II gives an overview over related work. Section III details the framework used to perform the test generation procedure. In Section IV the Validity Checker Module is explained and an extended classification of faults is presented. Finally, in Section V the case study and experimental results obtained are described. Conclusions are drawn in Section VI.

II. Related Work

The principle of SBST is to run functional test patterns, based on the processor instruction set, i.e., exploiting processor resources to test the processor itself [2]. It consists in forcing the processor to execute a sequence of instructions deliberately designed to thoroughly excite all possible faults and propagate the fault effects to the primary outputs of the circuit. This sequence of instructions is the so-called *test program*. Functional tests do not require circuit modifications and may offer good defect coverage, since they are executed at speed. Moreover, they can be performed both at the end of the production process, and during the operational phase (e.g., for periodical on-line testing).

Several approaches can be found in the literature resorting to manual, random, evolutionary algorithms, and hybrid techniques to generate test programs suitable for SBST. A conclusive summary of recent work on software-based self-testing of microprocessors is given in [2]. Also formal methods, particularly suitable for automation, were proposed to this end. In [4] a method is introduced, which generates test sequences at module level. Then, the Cadence SMV model checker is used to map these sequences to instruction sequences, which propagate the fault to a primary output. In [12] precomputed test patterns for modules are used. The circuit is abstracted to RT level and a SAT solver is used to justify and propagate the test patterns. A graph model for the behavior of a pipelined processor is proposed in [13], which constrains the test pattern generation for path delay faults. In [14] an automatic test program generation method is proposed with executing-trace-based constraint extraction for embedded processors, which facilitate structural test generation with constraints at gate level, and automatic test instruction generation (ATIG) also for hidden control logic.

Even if they all provide valid and usable test programs, none of them targets *small delay defects* (SDDs). Formerly, a SDD on a circuit would not necessarily cause the circuit to fail, because of the low operating frequencies. However, with operating frequencies getting always higher, the once negligible delay can become significant and cause failures. In the past few years tests targeting SDDs have gained importance. Mainly, they use the structural approach, resorting to automatic test pattern generation methods. Commercial tools already offer interesting options [15] [16] [17]. Different approaches can be found in the literature, for example [18], targeting industrial benchmarks, and the one in [19], selecting the most efficient patterns from timing-unaware ATPG tools and working on IWLS 2005 benchmarks circuits. Approaches for the identification of the longest paths through a fault site, which are crucial for SDD tests, are proposed in [20] and [21].

Nevertheless, at speed functional test, i.e., using functional patterns to test the chips at the target operating frequency, has the desired advantage of avoiding the over-testing problem (classifying defect-free chips as failing ones), thus raising the yield.

The aim of this work is the automatic generation of functional tests targeting small-delay faults in microprocessors. Our approach directly works on the gate-level model of the circuit, i.e., it does not require abstraction of the circuit. Additionally, it only requires a minor manual effort to model the processor's functional constraints. It forces the ATPG engine to produce valid test programs, taking into account the special constraints on the sequence of values to be applied to the input signals of the processor.

III. ATPG framework

In this section we describe an ATPG framework for small-delay faults in sequential, non-scan circuits [6]. Other fault models like stuck-at and transition faults are also implemented, but are not the focus of this work. The framework utilizes a bounded model checker with Craig interpolation [22] to generate a test sequence for a given small-delay fault or to prove that no such sequence exists. We also introduce a new step in the original ATPG flow in order to increase the number of detectable faults.

A. Bounded Model Checking with Craig Interpolation

In general, BMC is employed to calculate a trace of length k from an initial state I_0 with a transition relation $T_{i,i+1}$ to a target property P_k

$$BMC_k = I_0 \wedge T_{0,1} \wedge ... \wedge T_{k-1,k} \wedge P_k \quad (1)$$

$T_{i,i+1}$ defines the progress of the system from timeframe i to $i+1$, whereas P_k specifies the property to be verified. A classical BMC approach starts from $k = 0$ and checks the satisfiability of BMC_k for each value of k. The algorithm terminates when BMC_k is satisfied or k reaches a user-defined bound. In order to prove that a property P is not satisfiable for all k, k has to be increased until no more new states can be reached. In general, this requires impractical values for k and therefore leads to long runtimes.

One approach for a more efficient fix point computation was proposed by McMillan [23], which utilizes Craig interpolants [24]. The interpolants facilitate an over-approximation of the reachable system states and thereby significantly reduce the number of iterations required to prove the unsatisfiability of the target property.

B. ATPG framework - Preliminaries

In order to find a test pattern sequence for a small-delay fault, the necessary constraints for the sensitization and propagtion of the fault have to be specified as a

BMC problem instance. This instance is then passed to the employed BMC solver. If the solver returns a solution, the found test pattern sequence can be extracted. In case of unsatisfiability it has been proven that the fault is untestable.

The framework requires a circuit given as a gate-level netlist and a fault list as inputs. Before starting the test pattern generation, the circuit has to be in a defined state, i.e., all flip-flop values have to be known. Therefore, an initialization sequence is computed by creating a BMC problem instance with an initial all-X state of the flip-flops. The Tseitin-Encoding [25] of the circuit constitutes the transition relation. As the target property we require all flip-flops to be '0' or '1'. The state after applying the initialization sequence is used as the starting point for the test pattern generation. It is possible that no initialization sequence exists, for example because a register is assumed to be in a certain state after the power-up of the circuit. The solver may also abort with a timeout, if a solution exists but requires a high unrolling depth of the circuit. If no initialization sequence could be found, a functional circuit state is additionally required as an input.

In the first step of the test pattern generation the ATPG framework determines for each fault whether its sensitization and propagation are structurally possible. In order to check the untestability it assumes that all flip-flops are fully controllable and observable and tries to find a test pattern pair which sensitizes the fault and propagates the fault effect to a primary output or a flip-flop. This problem is denoted as a SAT formula and computed by a SAT solver. In case of unsatisfiability the fault is not considered anymore, as it also will not be possible to sensitize or propagate it from any functional circuit state.

The second step of the test pattern generation comprises the sensitization of the fault and the propagation of the fault effect to a primary output or a flip-flop. A BMC problem instance is generated, which uses the state after the application of the initialization sequence as the initial state and the Tseitin-Encoding of the circuit as the transition relation. The target property is specified by the SAT formula from the first step of the pattern generation. Consequently, the BMC solver tries to reach a circuit state where two consecutive test patterns can be applied, which sensitize the fault and propagate it to a primary or secondary output. We denote the resulting pattern sequence as the *path sensitization sequence*. Note that the sensitized paths are not precomputed, but directly extracted from the solver solution. By executing a binary search over the length of all paths through a target gate, the longest sensitizable path can be determined. If the described problem is unsatisfiable, it has been proven that it is not possible to test the fault from a functional circuit state.

If the sensitized path ends at a primary output, a test pattern sequence for the examined fault has been found. If the path ends at a secondary output (i.e., an input of a flip-flop) the fault effect still has to become visible at an output of the circuit. The third step of the test pattern generation is then required to compute a *fault propagation sequence*. A BMC problem instance is generated, which utilizes the last state of the path sensitization sequence as the initial state. In this state the fault effect is stored in one flip-flop of the circuit. The target property requires that this fault effect has to become visible at least at one primary output.

C. Propagation-aware path sensitization

The described ATPG process allows to generate a test sequence for a high percentage of faults, but it comprises one weakness: the path sensitization step does not consider the further propagation of the fault. This can lead to system states, in which a fault effect is latched, but cannot be propagated any further. For that reason we introduce an intermediate step, which realizes a propagation-aware path sensitization. After the computation of a path by the path sensitization step, we generate a BMC instance whose initial state is the state after the application of the initialization sequence. In the transition relation we require the sensitization of the computed sensitized path. Then all flip-flops which are reachable from the path end node are determined. Based on a precomputed heuristic, the flip-flops are chosen, which have the highest probability to propagate a fault effect to a primary output. In the target property we then require to not sensitize the computed path only, but to propagate its fault effect further to one or more of the selected flip-flops. If the solver finds a solution, the final state is passed to the fault propagation step. The intuition is that this final state is better suited for the propagation of the fault to a primary output. If the fault propagation step fails, the propagation-aware path sensitization is repeated. Now, not only the flip-flops which are reachable from the path end node are considered for the target property, but also the flip-flops reachable from the flip-flops sensitized in the previous propagation-aware path sensitization step. This procedure is repeated until a fault propagation sequence is found, the fault cannot be propagated further or a user-definded upper bound is reached.

IV. Constraining the inputs

The framework described in Section III allows to generate a test pattern sequence for a given circuit and fault or to prove that no such sequence exists. But the generated patterns can span over the complete input space, i.e., each possible sequence of input combinations can be applied to the primary inputs. In order to generate only functional input patterns, constraints have to be added, which prohibit invalid solutions. In the following we present our approach for the specification of constraints corresponding to valid functional input sequences.

A. Validity Checker

In order to forbid certain input values explicitly the SAT formulas and BMC problem instances from Section III-B could be modified accordingly. However, specifying the desired constraints with SAT clauses is tedious and counterintuitive, particularly for complex constraints. Moreover, the ATPG framework would have to be extended for each circuit so as to generate problem instances with the specified constraints.

Therefore, we propose the use of a *Validity Checker Module (VCM)*. This VCM can be specified in a hardware

description language like VHDL or Verilog. Its purpose is to formalize the valid input space for a circuit, i.e., to model the environment the circuit is embedded in. In this work we will focus on a VCM for a microprocessor, but the general approach is applicable for other circuit types as well.

The usual inputs of a microprocessor encompass a reset signal, interrupt signals, data busses and several signals for communication with memories and other components. A reset signal is typically activated only once after power-up and is then required to stay inactive. Interrupt signals usually appear with a certain distance of time. Finally, when loading an instruction, only valid instructions can be applied to the corresponding input bus of a processor. If these requirements are not considered during the generation of the test patterns, patterns will be produced, which cannot be applied in the functional test scenario.

A VCM is basically a circuit itself, which is located in parallel with the circuit under test (CUT). Its inputs are all the primary inputs and internal signals of the CUT, which are required to specify the desired constraints. For each constraint a validity output is added to the VCM, which is set to '1', if the constraint is satisfied and otherwise to '0'. Consequently, each output of the VCM corresponds to the validity of one constraint.

The specification of the constraints requires some engineering knowledge about the microprocessor and the system it is embedded in. For example, the complete instruction set has to be defined in the VCM in order to produce a validity signal for correct instructions. The use of internal signals of the microprocessor enables us to specify constraints, which also consider the internal state of the circuit. For instance, in a Von-Neumann-based processor architecture it is not possible to distinguish the load of an instruction from the load of a data word without the knowledge about the current internal state. Consequently, the required signals are transformed into primary outputs and are therefore accessible for the VCM.

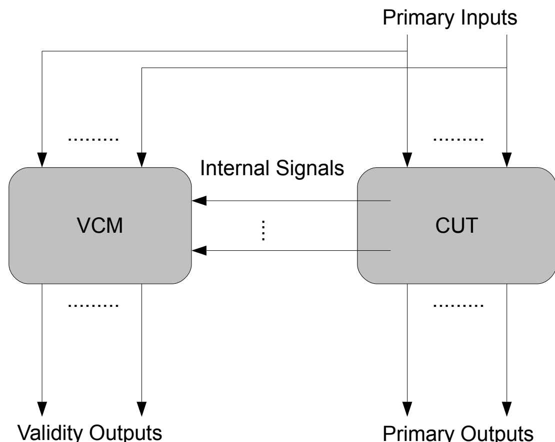

Figure 1. Concept of the Validity Checker Module

After the VCM has been specified it is synthesized to a gate-level netlist. Then, the VCM and the CUT netlists are combined properly (Figure 1) to form the final input netlist for the ATPG framework. The BMC problem instance generation, described in Section III-B, has to be modified in order to produce correct test pattern sequences. The transition relation now encompasses the Tseitin-Encoding of the CUT and the VCM. For each validity output of the VCM we add a clause to the transition relation, which forces the output to be always '1'. This requires the BMC solver to produce only test sequences, where all specified constraints are satisfied in each timeframe. In general, a satisfiable problem instance can become unsatisfiable through the addition of constraints. For that reason we propose the use of a VCM with several validity outputs. By removing the corresponding clauses from the transition relation we can disable one or several constraints in each step of the test pattern generation. This enables a user to adapt the constraints to the individual requirements of the test infrastructure of the CUT.

It is important to note that the VCM and the addition of primary outputs to the CUT are only required for the transformation of the constraints from VHDL code to clauses, which enables the functional test pattern generation. The final circuit does not have to be changed in any way.

B. Fault classification

The described approach facilitates the classification of faults into several categories.

The described ATPG framework can identify *structurally untestable* and *sequentially untestable* faults. The former comprises faults, for which no test pattern pair exists. For the latter faults there is at least one test pattern pair, but its application requires a system state, which cannot be reached starting from a functional initial state.

The utilization of the VCM introduces *functionally untestable* faults. For these faults there exists no test sequence which complies with all constraints specified in the VCM. However, they could be tested by removing one or more of the constraints.

V. Experimental Results

A. Experimental Setup

In our experiments we investigated all modules of the miniMIPS processor [11]. The processor was synthesized to a 18,279 gates netlist with Synopsys Design Vision using an in-house developed library. A small-delay fault for each gate was added to the fault list.

The primary inputs of the miniMIPS include a reset signal, a hardware interrupt signal, a memory acknowledgement signal and a 32 bit data bus. All of them are necessary inputs for the VCM. Additionally, we require the value of a register *currentState*, whose value reveals whether the processor is fetching an instruction or loading a data word. We then specify the following four constraints in the VCM.

The reset signal definitely needs to be set in order to initialize the processor, but we do not want it to be set afterwards, as setting the reset signal at a certain clock cycle would be hard to control in a functional test environment.

Table I
Detailed fault classification for coprocessor

	struc./seq. unt.	func. unt.	func. test.	abort
NoCon	200	0	872	8
Inst	200	3	863	14
Mem	200	3	852	25
Irt	200	39	821	20
All	200	39	819	22

Therefore, we allow the reset signal to be active immediately after power-up for an arbitrary amount of time and then require it to stay inactive. For similar reasons we force the hardware interrupt signal to be always inactive. The memory acknowledgement signal is required to be always active, which corresponds to a memory with a response time of one clock cycle.

Finally, we want the processor to load only valid instructions or abitrary data words. Thus we specify the complete instruction set of the miniMIPS in the VCM and utilize the register currentState for distinguishing between the load of an instruction or a data word.

These constraints have been defined with the purpose of validating the effectiveness of the approach. Other constraints could be used depending on the required characteristics of the generated test patterns.

The test pattern generation with the described constraints yields a functional test pattern sequence consisting only of valid instructions or data words. Based on this a real test program may be easily obtained with limited additional effort. To do so, each instruction and data word has to be mapped to a memory address. In particular, load and branch instructions have to be taken care of, as they could target arbitrary memory addresses. We are currently extending the VCM to generate a test sequence, which directly corresponds to a correct memory mapping. Moreover, constraints will be added which require the fault effects not only to be visible on a primary output, but to be written in the memory.

B. Experimental Evaluation

In the experiments we targeted a small-delay test for the longest non-robustly sensitizable path through each gate. First we evaluated the coprocessor (*syscop*, 1080 gates) in detail by adding the described constraints one after the other. This way we could determine which constraint introduced functionally untestable faults. Then we performed the test pattern generation process for all 18,279 gates of the miniMIPS processor. We executed a run with unconstrained inputs and a run with all of the described constraints. All experiments were performed on an Intel Xenon processor running at 3.3 GHz. The synthesized VCM consists of 1 flip-flop and 97 gates. It requires all inputs of the miniMIPS and the value of *currentState* as inputs and has four outputs for the described four constraints. The same VCM was used for all experiments. In general the size of a VCM is determined only by the complexity of the specified constraints.

Table I shows the detailed evaluation of the coprocessor. The first column encompasses all structurally and sequentially untestable faults. First no constraints are applied (*NoCon*) and thus no functionally untestable faults are present. Then the described constraints are added one by one in the following order: valid instructions (*Inst*), valid memory acknowledgement (*Mem*), valid interrupt (*Irt*) and valid reset (*All*). The last row (*All*) presents the results when considering all constraints. It can be seen that the majority of functionally untestable faults is introduced by forcing the interrupt signal to be always inactive. This is reasonable, as the coprocessor is responsible for the interrupt handling.

Table II shows the obtained results for all miniMIPS modules. Column *untestable* includes the number of all provably untestable faults. Column *func. testable* gives the number of faults for which a functional test sequence could be generated.

The column *abort* in Table II lists the faults which could not be classified. We executed a binary search by length through all sensitizable paths and aborted the current fault after an user-defined upper bound in each step of the search, if no solution was found. For all of these aborted faults we were able to sensitize a path and propagate the fault effect to a flip-flop, but could not find an input sequence, which propagated the fault effect to a primary output. Moreover the applied solver was not able to prove the unsatisfiability of the propagation of these faults.

Column *avg. cycles* lists the average length of the test sequences for one fault expressed in number of clock cycles (cc). The sequences range from 2 to 25 clock cycles. The initialization sequence for the miniMIPS required 7 clock cycles. Note that in general the number of pipeline stages is not a sufficient upper bound for the number of required circuit unrollings until a fault is proven to be untestable. For example, testing a gate in the branch prediction unit may require several branch instructions until a proper system state is reached.

The runtime for generating a test sequence for one fault ranges from several seconds up to over an hour. The second last column presents the average runtime for one fault in minutes (m). These runtimes seems high, but can be justified by the complexity of sequential ATPG and the small-delay fault model, whose combination requires a thorough search through a large search space. Furthermore, the utilized ATPG framework is, to the best of our knowledge, the only ATPG for small-delay faults able to generate functional test programs for a real-sized processor. Finally, as all faults are processed one by one, the ATPG procedure can be easily parallelized.

The last column indicates the *Fault Efficiency (FE)* in percent (i.e., the ratio between detected and testable faults). It can be seen that our approach performs very well on the miniMIPS, as the numbers are comparable to stuck-at ATPG [26] and we reach an overall fault efficiency of 97.33 %. In particular, we would like to point out the results concerning the data forwarding unit (*renvoi*) and the branch prediction unit (*predict*). Both of these modules require very specific instruction sequences in order to trigger their functionality and are therefore considered hard to test because they imply hidden control logic.

Without the application of the propagation-aware path sensitization 2103 aborts occurred for the whole miniMIPS.

Table II
TEST GENERATION RESULTS FOR MINIMIPS

	# gates	untestable	func. testable	abort	avg. cycles [cc]	avg. runtime [m]	FE [%]
pf	348	0	348	0	10.01	0.66	100
ei	268	0	263	5	11.25	13.55	98.1
di	1180	74	1066	40	9.54	15.63	96.4
ex	3812	144	3490	178	11.93	3.18	95.2
mem	468	124	304	40	12.92	2.78	88.4
renvoi	551	40	507	4	11.73	6.41	99.2
banc	6805	0	6642	163	10.90	1.34	97.6
syscop	1080	239	819	22	13.19	0.37	97.4
bus_ctrl	346	46	296	4	8.19	5.68	98.7
predict	3421	9	3398	14	15.00	3.64	99.6
total	18279	676	17133	470	11.95	3.46	97.3

This number could be reduced down to 470 aborts with the method introduced in Section III-C. The overall number of faults for which a functional test sequence could be found increased by 9.53 % from 15500 to 17133.

Finally, the experimental results show that 3.61 % of the faults are either structurally, sequentially or functionally untestable. For all of these faults it has been formally proven that they cannot be tested under the specified constraints, which enabled the computation of a fault efficiency ratio for the circuit.

VI. Conclusions

We presented the first fully automated approach to functional microprocessor test generation for small-delay faults. Our framework allows the specification of constraints related to functional requirements and the proof of untestability under these constraints. We verified our approach by evaluating a pipelined microprocessor and provided experimental results, which demonstrate the feasibility and relevance of our work.

In the future we will investigate extensions of the described VCM, which shall enable the automated generation of test programs suitable for the on-line software-based self-test of a microprocessor. This implies taking into account the memory mapping and including some constraints regarding the observability methods. Furthermore, we will explore other formal verification techniques in order to classify a higher number of faults.

Acknowledgements

Parts of this work were supported by the German Research Foundation (DFG) under grant GRK 1103.

References

[1] P. C. Maxwell, R. C. Aitken, K. R. Kollitz, and A. C. Brown, "IDDQ and AC scan: The war against unmodelled defects," in *International Test Conference*, pp. 250–258, 1996.

[2] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda, "Microprocessor software-based self-testing," *IEEE Design and Test of Computers*, pp. 4–19, 2010.

[3] S. M. Thatte and J. A. Abraham, "Test generation for microprocessors," *IEEE Transactions on Computers*, pp. 429–441, 1980.

[4] S. Gurumurthy, S. Vasudevan, and J. A. Abraham, "Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor," in *IEEE International Test Conference (ITC)*, pp. 1–9, 2006.

[5] S. Gurumurthy, R. Vemu, J. A. Abraham, and D. G. Saab, "Automatic generation of instructions to robustly test delay defects in processors," in *IEEE European Test Symposium*, pp. 173–178, 2007.

[6] M. Sauer, S. Kupferschmid, A. Czutro, I. Polian, S. Reddy, and B. Becker, "Functional test of small-delay faults using SAT and craig interpolation," in *IEEE International Test Conference (ITC)*, pp. 1–8, 2012.

[7] R. Mattiuzzo, D. Appello, and C. Allsup, "Small-delay-defect testing," *Test & Measurement World*, pp. 37–41, 2009.

[8] F. Corno, M. Sonza Reorda, and G. Squillero, "RT-level ITC'99 benchmarks and first ATPG results," *IEEE Design & Test of Computers, Vol. 17, Issue 3*, pp. 4–53, 2010.

[9] P. Bernardi, M. Bonazza, E. Sanchez, M. Sonza Reorda, and O. Ballan, "On-line functionally untestable fault identification in embedded processor cores," in *Design, Automation & Test in Europe*, pp. 1462–1467, 2013.

[10] R. S. Tupuri and J. A. Abraham, "A novel functional test generation method for processors using commercial ATPG," in *IEEE International Test Conference (ITC)*, pp. 743–752, 1997.

[11] *miniMIPS*. http://opencores.org/project,minimips.

[12] L. Lingappan and N. K. Jha, "Satisfiability-based automatic test program generation and design for testability for microprocessors," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 518–530, 2007.

[13] V. Singh, M. Inoue, K. K. Saluja, and H. Fujiwara, "Instruction-based self-testing of delay faults in pipelined processors," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1203–1215, 2006.

[14] Y. Zhang, H. Li, and X. Li, "Automatic test program generation using executing-trace-based constraint extraction for embedded processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1220–1233, 2013.

[15] M. Graphics, "At-speed and advanced fault models for achieving high quality test." White Paper, December 2009.

[16] Cadence, "Encounter True-Time ATPG data sheet."

[17] Synopsys, "TetraMAX ATPG data sheet."

[18] S. K. Goel, N. Devta-Prasanna, and R. P. Turakhia, "Effective and efficient test pattern generation for small delay defect," in *IEEE VLSI Test Symposium*, pp. 111–116, 2009.

[19] M. Yilmaz, K. Chakrabarty, and M. Tehranipoor, "Test-pattern selection for screening small-delay defects in very-deep submicrometer integrated circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 760–773, 2010.

[20] X. Lu, Z. Li, W. Qiu, D. M. H. Walker, and W. Shi, "Longest path selection for delay test under process variation," 2004.

[21] M. Sauer, J. Jiang, A. Czutro, I. Polian, and B. Becker, "Efficient sat-based search for longest sensitisable paths," in *Asian Test Symposium (ATS)*, pp. 108–113, 2011.

[22] S. Kupferschmid, M. Lewis, T. Schubert, and B. Becker, "Incremental preprocessing methods for use in BMC," *Formal Methods in System Design*, pp. 1–20, 2011.

[23] K. L. McMillan, "Interpolation and SAT-based model checking," in *Intl Conference Computer Aided Verification*, pp. 1–13, 2003.

[24] W. Craig, "Linear reasoning: A new form of the Herbrand-Gentzen theorem," *Journal of Symbolic Logic*, pp. 250–268, 1957.

[25] G. S. Tseitin, "On the complexity of derivation in propositional calculus," *Studies in Constructive Mathematics and Mathematical Logics*, 1968.

[26] M. Hatzimihail, M. Psarakis, D. Gizopoulos, and A. Paschalis, "A methodology for detecting performance faults in microprocessors via performance monitoring hardware," in *IEEE International Test Conference (ITC)*, pp. 1–10, 2007.