A Flexible Framework for the Automatic Generation of SBST Programs

A Flexible Framework for the Automatic Generation of SBST Programs

Andreas Riefert, *Member, IEEE*, Riccardo Cantoro, *Member, IEEE*, Matthias Sauer, *Member, IEEE*, Matteo Sonza Reorda, *Fellow, IEEE*, and Bernd Becker, *Fellow, IEEE*

***Abstract*— Software-based self-test (SBST) techniques are used to test processors and processor cores against permanent faults introduced by the manufacturing process or to perform in-field test in safety-critical applications. However, the generation of an SBST program is usually associated with high costs as it requires significant manual effort of a skilled engineer with in-depth knowledge about the processor under test. In this paper, we propose an approach for the automatic generation of SBST programs. First, we detail an automatic test pattern generation (ATPG) framework for the generation of functional test sequences. Second, we describe the extension of this framework with the concept of a validity checker module (VCM), which allows the specification of constraints with regard to the generated sequences. Third, we use the VCM to express typical constraints that exist when SBST is adopted for in-field test. In our experimental results, we evaluate the proposed approach with a microprocessor without interlocked pipeline stages (MIPS)-like microprocessor. The results show that the proposed method is the first approach able to automatically generate SBST programs for both end-of-manufacturing and in-field test whose fault efficiency is superior to those produced by state-of-the-art manual approaches.**

***Index Terms*— Automatic software-based self-test (SBST), functional ATPG, microprocessor test, SBST, SBST for in-field test.**

I. Introduction

TESTING microprocessors for permanent faults emerging during the manufacturing process or during the operational phase is a complex task, regardless whether the processor is a standalone device or a core within a system on a chip. The most suitable solution depends on the specific scenario and on the specific technology. In some cases, design for testability (DfT) perfectly fits the requirements. In other cases, it is necessary to complement DfT solutions with other techniques, e.g., functional test.[1] For example, the adoption of some recent technologies makes the test of delay defects particularly important, which sometimes can hardly be detected using traditional DfT solutions (e.g., scan). There are also cases, in which DfT is simply not feasible. For example, when the in-field test of a board or system must be developed by a system company, sufficient information about any DfT structure is often not made available by the device providers, and functional test becomes the only feasible solution.

When processors are considered, functional test typically takes the form of software-based self-test (SBST) [1]: the processor is forced to execute a given test program, and faults are detected by looking at the results produced by the program (e.g., in terms of values written in memory). Unfortunately, functional test suffered in the past from the fact that the complexity for generating suitable test stimuli may be prohibitively high. When considering a processor, the task of developing suitable test programs was mainly performed in a manual manner, thus raising significant concerns in terms of required cost and time.

Several functional test approaches have been proposed for microprocessors over the last three decades [2]–[5]. In the last decade, functional approaches have also been increasingly adopted by industry [6]–[8]. In the recent years, the growing adoption of processor-based systems in safety-critical applications significantly increased the need for effective solutions for their in-field test. Furthermore, the emergence of standards and regulations (e.g., IEC 61508 for industrial safety-related systems, ISO 26262 for automotive applications, and DO-254 for avionics) further pushed industries and researchers to focus on the in-field test of such systems. As a result, several works dealt with the development of techniques for writing effective test programs for whole processors or for some popular components within a processor. For example, the method described in [9] allows to write test programs able to effectively test caches, while in [10], an approach was proposed that focuses on the test of branch prediction units. In [11], a technique is described for the test of memory management units. In addition, functional ATPG tools based on formal methods, such as satisfiability (SAT) and bounded model checking (BMC), have been proposed [12]–[14]. However, these approaches struggled with the complexity of handling a complete processor within a BMC formula. For example, in [12], this is handled by generating module level tests and mapping them to instructions. This is often not possible as module level tests may require nonfunctional system states. While often stuck-at faults are considered, also methods for the functional test of delay faults have been developed [15], [16]. Furthermore, functional methods for detecting

Manuscript received August 7, 2015; revised October 26, 2015 and January 25, 2016; accepted March 3, 2016. The work of B. Becker was supported in part by the Deutsche Forschungsgemeinschaft, in part by the Federal Ministry of Education and Research, and in part by the Industry.

A. Riefert, M. Sauer, and B. Becker are with the Department of Computer Science, University of Freiburg, Freiburg im Breisgau 79110, Germany (e-mail: riefert@informatik.uni-freiburg.de; sauerm@informatik.uni-freiburg.de; becker@informatik.uni-freiburg.de).

R. Cantoro and M. Sonza Reorda are with the Department of Control and Computer Engineering, Politecnico di Torino, Turin 10129, Italy (e-mail: riccardo.cantoro@polito.it; matteo.sonzareorda@polito.it).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2016.2538800

[1] In this paper, we denote a test, which can only control the functional inputs and can only observe the functional outputs as a functional test.

faults, which do not affect the correct functionality of a processor but degrade its performance, have been investigated [17]. In [18], an approach was proposed for the generation of SBST programs, which is mainly based on the execution of a large number of blocks of random instructions (>1 million). However, it does not succeed in achieving high fault coverage when only the memory content can be observed after SBST program execution. In [19], an approach to SBST program generation is presented, which is based on the creation of an abstract simulation model of the processor under test. Random training programs are used to create a mapping from processor inputs and outputs to inner module ports. This mapping is used in conjunction with constrained structural ATPG in order to derive an SBST program. Furthermore, special test routines for hidden control logic as data forwarding and branch prediction are proposed. While this approach is able to achieve high fault coverage, it requires in-depth knowledge of the processor for creating a simulation model and test routines for hard-to-test logic.

At the same time, some companies providing microcontrollers for safety-critical applications started to deliver libraries of software procedures that, when executed by the processor, guarantee the achievement of specified fault coverage. These libraries are integrated by the system company into the application software, and their execution is properly triggered depending on the test and reliability specifications. Clearly, their development represents a significant cost, which could be reduced if at least some parts could be automatically generated, starting from the processor netlist.

While the discussed works provide generalized approaches to (semi-)manual test program generation or automatically generated functional test sequences (TSs), none of them allows a user to model the constraints of an SBST program for an arbitrary processor and test environment on an abstract level. For example, a constraint could require to only apply valid instructions. Especially when targeting in-field SBST, restrictive constraints are imposed, e.g., the memory area available for the test code and data may be limited [20], some input signals may hardly be controllable (e.g., reset), and only the final content of the memory can be observed. Hence, the generation of the corresponding test program is significantly more complex than for end-of-manufacturing test. Previous works also showed that during in-field test, a higher number of faults become untestable [21]. Thus, a general approach to model the described constraints is mandatory to achieve the same degree of automatism for SBST generation as it is common for scan test patterns in (commercial) ATPG tools. Furthermore, an effective identification of untestable faults under the specific constraints is necessary to comply with the regulations in the application field of the processor.

The first contribution of this paper is to describe a method, based on formal techniques, which is able to automatically generate suitable test programs for mid-sized pipelined processors, such as those that are often used in microcontrollers. These test programs can be used for both end-of-manufacturing and in-field test. The method introduces several optimizations with respect to previous attempts in the area, which can reduce the computational effort and maximize the achieved fault coverage. It is worth mentioning that the method is able not only to generate a test program with high fault coverage, but also to prove the untestability of faults. The employed solving engine applies unbounded model checking for the first time in the scope of SBST generation. This enables to handle a complete mid-sized processor at gate level within the solving engine, which has not been possible with other approaches [12]. The employment of abstraction techniques in combination with the powerful solving engine will allow to handle even larger designs.

The second contribution of this paper specifically focuses on in-field test. This kind of test requires launching the execution of suitable procedures either at the power-ON or during the idle slots of the application. By looking at the produced results, the system can detect possible faults affecting the processor. We list constraints, which often exist when performing in-field SBST test of a processor, and propose a method, which allows to use the optimized ATPG algorithm mentioned before in combination with these constraints. In practice, the method allows the test engineer to specify the constraints existing in a given environment, and thereby forcing the ATPG algorithm to generate a test program matching them. As a result, this paper is the first to propose a method able to automatically generate effective test programs to be used for in-field SBST test of a processor. A major advantage of this method lies in the fact that it is also able to identify faults, which cannot be tested, when constraints are introduced. Experimental results gathered on an MIPS-like processor show the feasibility and the effectiveness of the proposed solutions.

This paper is an extension of [22], where the combination of specified constraints with the described ATPG was introduced, and of [23], where exemplary constraints for SBST were developed and evaluated. In this paper, we improve the runtime of the functional test generation engine by extracting and reusing knowledge gained during the test generation process and by implementing a heuristic for the reduction of aborts. We give a detailed and generalized description of typical in-field SBST constraints and their integration into our framework. Finally, we present extensive experimental results with significantly improved performance compared with the previous works.

The rest of this paper is organized as follows. Section II introduces two formal tools, which are used in this paper. Section III details the functional ATPG algorithm and its optimizations. In Section IV, the interface for the specification of constraints is described. Section V introduces the considered constraints, which are present in a typical SBST scenario. Finally, Section VI shows the experimental results and Section VII concludes this paper.

II. Preliminaries

In this section, we give an introduction to two formal techniques, which are employed in this paper. First, we detail BMC with Craig interpolation, which allows to determine the (non-)existence of a trace from an initial state to a target state. Thus, the combination of BMC with Craig interpolation enables unbounded model checking. Second, we give a brief introduction to maximum SAT (MAX-SAT), which is a generalization of the Boolean SAT problem.

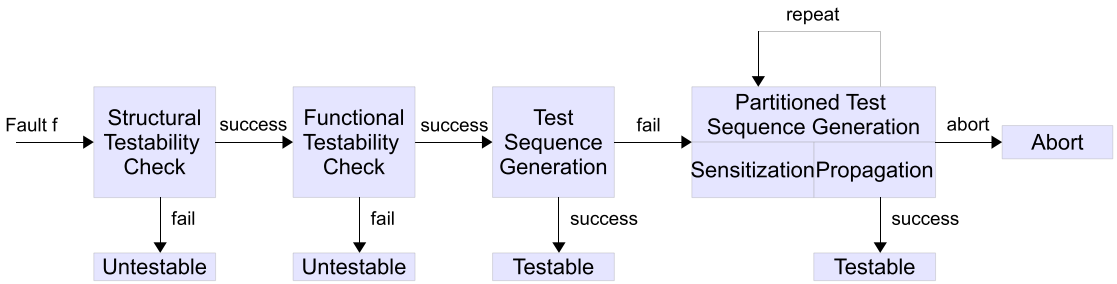

Fig. 1. Processing steps for a fault.

A. Bounded Model Checking With Craig Interpolation

A classical BMC solver tries to solve a formula, which is defined by an initial state I_0, a transition relation $T_{i,i+1}$, and a target property P_k

$$\mathrm{BMC}_k = I_0 \wedge T_{0,1} \wedge \cdots \wedge T_{k-1,k} \wedge P_k. \quad (1)$$

$T_{i,i+1}$ defines the progress of the system from time frame i to $i+1$, whereas P_k specifies the property to be verified. Starting with $k = 0$, the solver searches for a solution, which satisfies the target property or proves that the target property cannot be satisfied within k steps. k is increased stepwise until a solution is found or no new system states can be reached. In general, the latter case requires very large values for k, which are not feasible in practice.

However, several approaches for a more efficient unreachability proof exist. In this paper, the solver Craig Interpolation prover (CIP) is employed [24], which uses Craig interpolants to over approximate the reachable system states within each step. This in many cases allows to prove effectively that a target property will not be satisfied for arbitrary values of k, which corresponds to unbounded model checking.

B. MAX-SAT

The underlying engine of our algorithm is an efficient SAT-based maximization tool, which solves the so-called MAX-SAT. An ordinary SAT problem consists of a number of clauses. An SAT solver has to find a solution, which satisfies each single clause and thus satisfies the SAT formula. If no such solution exists, the formula is unsatisfiable. An MAX-SAT problem is a generalization of SAT and its target is to determine the maximum number of clauses that can be satisfied simultaneously.

In the following, we provide a brief overview of the employed MAX-SAT solver [25]. The solver distinguishes between two types of clauses, namely, hard clauses and soft clauses. A valid solution has to satisfy all hard clauses and the maximum number of soft clauses. This problem is solved by incrementally calling an SAT solver. In order to transform the original problem, consisting of hard and soft clauses, into a standard SAT problem, which only consists of hard clauses, the formula has to be modified. For this purpose, a bitonic sorting network is employed and encoded into the formula. This network can be viewed as a circuit with n inputs (corresponding to the soft clauses) and n outputs. Its function is to sort all 0s and 1s [each 0 (1) corresponding to an unsatisfied (a satisfied) soft clause] applied at the inputs to form a nondecreasing sequence. Alternatively, the network can be understood as a counter, which counts all applied 1s and outputs the result in a unary representation. This enables us to count the number of satisfied soft clauses and incrementally adjust the bounds for this number until the optimal solution is found.

To prevent excessive runtimes, the employed MAX-SAT solver works with a timeout. If the timeout is reached, the solver returns the best solution that it has found so far. Furthermore, the solver comprises a partial mode, which does not return the optimal solution but incrementally optimizes the blocks of soft clauses. The block size is a user-defined parameter. Small block sizes yield increased performance but, in general, produce worse results.

III. ATPG Framework for Functional Test Generation

In this section, we give a detailed description of the proposed functional ATPG framework, which is based on the solver introduced in Section II-A. While also other functional ATPG engines could be used, SAT-based approaches have shown to be very effective for targeting hard-to-detect faults and identifying untestable faults. The proof of structural or functional untestability for a considered fault allows to correctly classify it and reduces the overall abort ratio.

The ATPG framework starts with a fault list, which initially contains all faults. All structurally equivalent faults are then collapsed. The faults from the collapsed fault list are processed one after each other. If a TS could be generated for a fault, all yet untested faults, which are tested by this sequence, are removed from the fault list. Processing one fault consists of the steps shown in Fig. 1. These steps are described in Sections III-A–III-D. The structural testability check is done with an SAT solver; all remaining steps are based on the CIP solver. All CIP-based steps require an initial state I_0. We use the reset state,[2] when no TS has been found yet, or the state reached after applying the last pattern of the previously generated sequence. Finally, Section III-E describes an optimization of the framework, which extracts knowledge from successfully generated TSs and reuses this knowledge to speed up the further TS generation.

[2]The reset state is determined by calculating a synchronization sequence, which starts in the all-X state and brings the circuit into a well-defined reset state. Thus, the synchronization sequence can bring the circuit from each arbitrary state into the reset state.

This functional ATPG framework is the underlying engine for the automatic generation of an SBST program. When applying additional constraints for SBST (Section V) to the ATPG algorithm, the generation of a TS for a fault will correspond to finding an assembler code snippet, which tests this fault. By using the final state of the last sequence as the starting state for the next sequence, we ensure that all snippets can be consecutively connected to a single test program. Furthermore, the ability of the solver to prove unreachability will identify faults for which no system state can be reached from which the fault can be tested. Consequently, it is not possible to generate a code snippet, which will test this fault, i.e., the fault is proved to be untestable.

A. Structural Testability Check

The first step determines whether a fault is structurally testable. We denote a fault as structurally testable if it can be tested with full control over all primary inputs and secondary inputs (i.e., flip-flop outputs) and full observability of all primary outputs and secondary outputs (i.e., flip-flop inputs). Consequently, a structurally testable fault can be tested with a full-scan approach. In this step, we employ an SAT solver as only one unrolling of the circuit has to be considered. For the input cone of the fault, only the fault-free version of the circuit has to be encoded. The output cone has to be encoded in fault-free and faulty versions in order to determine the fault effect. By solving the corresponding formula, the SAT solver either finds a structural test for the fault or proves that no such test exists. In the latter case, the fault does not have to be considered further as it will also be untestable in the functional scenario. This step is reasonable as it can effectively identify structurally untestable faults with only one unrolling of the circuit.

B. Functional Testability Check

The second step evaluates whether a TS exists, which is able to sensitize the fault and propagate it to a primary or secondary output within the same cycle. For this purpose, a CIP formula has to be generated. For the transition relation $T_{i,i+1}$, the whole circuit is encoded in a fault-free version. In addition, the output cone of the fault location, specifying the faulty behavior, is encoded in the transition relation. In the target property P_k, we require a difference between the fault-free and the faulty output cone of the fault location. If the solver returns unsatisfiable, then the fault is functionally untestable as no circuit state can be reached, which sensitizes the fault and propagates it to an output. In combination with Footnote 2, we can also conclude that no such state can be reached from the initial state. Thus, there is no functional system state, which allows testing the fault. This check is reasonable as it only requires encoding the faulty output cone of the faulty circuit instead of the complete faulty circuit (which is required for the next steps) and is, therefore, a less complex problem. If the solver returns satisfiable, we proceed with the next step.

C. Test Sequence Generation

This step tries to generate a TS, which sensitizes the considered fault and propagates it to a primary output. For the transition relation of the CIP formula, we now have to encode the complete circuit in a fault-free and a faulty version. This is necessary as the fault effect may be propagated through several cycles and arbitrary parts of the circuit before it reaches a primary output. In the target property, we require a difference between the fault-free and the faulty circuit at a primary output. If the solver returns satisfiable, we can extract the TS from the solution. If the solver aborts due to a timeout, we proceed with the next step.

D. Partitioned Test Sequence Generation

If a fault is testable, but requires a long TS, the TS generation step may abort with a timeout, as a large number of circuit unrollings are required. Therefore, this step tries to generate a TS by partitioning the problem into two subproblems, namely, sensitization and propagation. The sensitization step sensitizes the fault and latches it into at least one flip-flop. Then the propagation step tries to propagate the latched fault effect to a primary output.

For the transition relation of the sensitization step, we encode the complete circuit in a fault-free and a faulty version. In the target property, we require the fault effect to be latched at least in one of a set of suitable flip-flops. The initial state of the propagation step is then given by the final state of the sensitization step, i.e., a circuit state where at least one flip-flop contains a fault effect. The transition relation also contains the encoding of the fault-free and the faulty circuit. In the target property, we require a difference between the fault-free and the faulty circuit at a primary output. The selection of suitable flip-flops is crucial for the success of this ATPG step. For this purpose, we precompute a heuristic for each flip-flop, which estimates the probability that a fault effect latched in this flip-flop can be propagated to a primary output. This heuristic is computed by choosing several random functional states. Then, for each flip-flop, a fault effect is inserted and a CIP formula is generated and solved, which tries to propagate this fault effect to a primary output. The partitioned TS generation is executed for several iterations until a solution is found or a user-defined bound is reached. In the first iteration, only flip-flops, which are structurally reachable from the fault location, are considered. If the fault effect could be latched into flip-flop F, then, in addition, all flip-flops, which are structurally reachable from F, are considered in the next iteration. Thus, the iterations of the partitioned TS generation will propagate the fault effect successively to varying flip-flops, until the fault effect is latched into a state, which allows its propagation to a primary output. If the propagation step succeeds, then the TS is extracted from the solutions of the sensitization and propagation steps. If the user-defined bound is reached, the fault is classified as aborted.

E. Fault Propagation Sequences

In this section, we will describe a modification of the TS generation step (Section III-C), which decreases the runtime by reusing knowledge gained from previously generated TSs.

A valid TS for a fault has to achieve two tasks. First, the fault has to be sensitized and latched into a register.

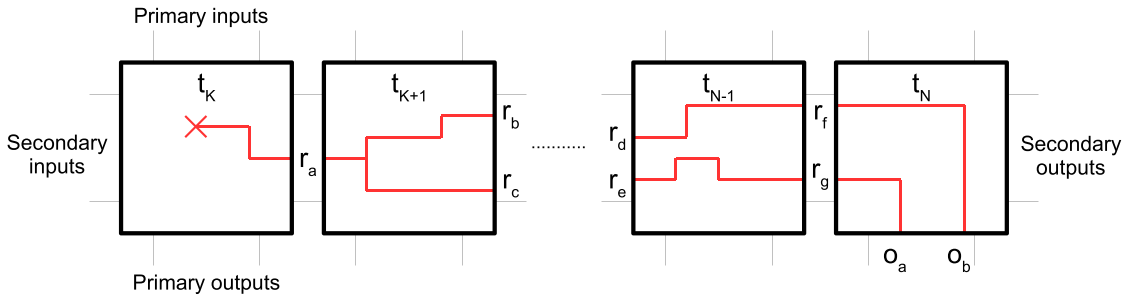

Fig. 2. Propagation sequence extraction.

Second, the faulty value of the register has to be propagated to an observable output of the circuit. While the sensitization is fault specific, the propagation of the latched register value is often fault independent. Therefore, we propose an approach that is based on a propagation sequence cache maintaining propagation sequences for each register r, which will allow the propagation of a value from r to a circuit output. These propagation sequences are extracted during the execution of the ATPG algorithm. Each successfully generated TS is used for extraction.

1) Propagation Sequence Extraction: The algorithm for the extraction of propagation sequences requires a TS for a fault F as an input. By fault simulation, we determine the traces over which the fault effect is propagated from the fault source to a primary output. Assume that TS consists of N clock cycles, and F is sensitized in cycle t_K. An example of such a sequence is shown in Fig. 2. The algorithm will start with the last clock cycle t_N. The set OD_N of all observable outputs, which show a fault effect in this cycle, is identified, i.e., o_a and o_b in the example. For each output $\mathrm{od}_N \in \mathrm{OD}_N$, we will then execute the following flow. The fault propagation path in the currently evaluated cycle t_N from od_N to an originating register id_N is identified, e.g., from o_a to r_g. Then an MAX-SAT formula (see Section II-B) is build. We use $01X$-encoding where an X is regarded as a don't care value. First, the input cone of this path is encoded. All side inputs of the path are required to maintain their noncontrolling value. All primary and secondary inputs in the input cone are required either to maintain their logic value or to be X. Finally, a maximization condition over the number of primary and secondary inputs being X is formulated. By solving this formula, the MAX-SAT solver will return a partially specified pattern PTS_N, where the $01X$-minimal number of primary and secondary inputs is specified, which still enables the fault propagation through the required path. This pattern then constitutes a fault propagation sequence with a length of one clock cycle starting at register id_N. If there is more than one originating register, the flow is repeated for all of them. Then, the next output od_N is evaluated.

When all $\mathrm{od}_N \in \mathrm{OD}_N$ values have been evaluated, the algorithm will continue with clock cycle t_{N-1}. Now, the set OD_{n-1} consists of the registers id_N (r_f and r_g in the example) from the previously evaluated cycle t_N. The elements of this set will be denoted by od_{N-1}. Similar to cycle t_N, for each od_{N-1}, the following flow is executed: The fault propagation path in cycle t_{N-1} from od_{N-1} to an originating register id_{N-1} is identified, e.g., from r_f to r_d. The MAX-SAT formula is built by first generating $01X$-encoding of the input cone of this path. In addition, the pattern PTS_N from cycle t_N has to be considered. The input cone of each register, whose value is not X in PTS_N, is also $01X$-encoded. Furthermore, these registers are required to have the same logic value as in PTS_N. The remaining formula is generated as described above. The solution of this MAX-SAT formula is a partially specified pattern PTS_{N-1} with the minimal number of specified primary and secondary inputs and enabled fault propagation through the desired path. Furthermore, the secondary outputs of this pattern are compatible with the secondary inputs of PTS_N. By concatenating PTS_{N-1} and PTS_N, we obtain a fault propagation sequence with a length of two clock cycles, which propagates a fault from the register id_{N-1} to the primary output od_N. An example for such a propagation sequence would be the path from r_d to r_f and then to o_b. Then, the next od_{N-1} is considered.

In general, there can be several propagation paths originating from a register r. In order to reduce the processing time, we determine whether an already computed partially specified pattern PTS for r also covers other propagation paths from r. If this is the case, these paths do not have to be considered.

The algorithm will evaluate all clock cycles until reaching t_K. During each cycle t_I, we extract partially specified patterns PTS_I and connect them with their corresponding patterns from the following time frames. This enables us to efficiently compute propagation sequences of arbitrary length. The fault propagation sequence extraction algorithm is applied to each generated TS during the ATPG flow.

2) Application of Propagation Sequences: The computed fault propagation sequences are integrated into the ATPG flow by modifying the TS generation step described in Section III-C. This modification is detailed in the following. We first determine which registers are structurally reachable from the considered fault over a certain amount of clock cycles. If no propagation sequences have been cached for any of these registers, the TS generation step is executed as usual. The amount of evaluated clock cycles is determined by the user. Furthermore, the user defines how many propagation sequences are chosen as targets. The selection algorithm will then start with the registers that are structurally reachable within one clock cycle and select one register and

its corresponding propagation sequence, which has the lowest number of specified secondary inputs. Then, the selection will proceed with the registers structurally reachable within two clock cycles. When the maximum number of considered clock cycles is reached, the selection process will start again with the registers reachable within one cycle. This process continues until the required number of propagation sequences is chosen. Next, a CIP formula is constructed, similar to Section III-C. For the transition relation, we also generate encoding of the fault-free and the faulty version of the complete circuit. The target property is now modified. It requires that the fault effect is latched into a system state, which is compatible with one of the targeted propagation sequences. This means that the fault effect has to be latched into the register, which is given by the propagation sequence, and the other register values have to match the values of the specified secondary inputs given by the propagation sequence. If such a state is reachable, we can construct the final TS for the considered fault by extracting the resulting sequence from the CIP formula solution and attaching the propagation sequence. We determine by fault simulation that the fault is detected by the final sequence. In general, it may happen that the desired propagation is hindered because of reconvergent fault effects. In this case, the generation of a TS with the propagation sequence cache has failed and we proceed with the partitioned TS generation step, thus trying to generate a TS without the cache.

F. Prediction of Aborts

The first experimental results showed that the list of aborted faults contained several blocks of faults, which consisted of structurally very similar faults. This structural similarity is usually due to several gates being located in the input cone of different flip-flops, which belong to the same multibit register (e.g., a 32-bit register). When considering such a block of structurally similar faults, the solver will try to solve repeatedly almost the same problem for each fault. Therefore, we implemented a heuristic, which identifies faults that are structurally similar to previously aborted faults in order to avoid unnecessary processing. Structural similarity of two faults is determined by comparing the port and the type of the gates affected by the two faults as well as the inputs, outputs, and (multibit) registers located in the input and output cone of the two fault sites. If all of these points are equal, the two faults are considered to be structurally similar. For the heuristic, all aborted faults since the last successfully generated TS are stored. When considering the next fault for ATPG, we determine whether this fault is structurally similar to a previously aborted fault. If this is the case, this fault is immediately classified as aborted without processing it.

IV. Validity Checker Module

The functional ATPG framework described in Section III assumes that the primary inputs of a circuit under test (CUT) can be set to arbitrary values at each clock cycle, and the primary outputs can be observed at each clock cycle. While this assumption is valid in a production test scenario, it is not realistic in an SBST environment. Usually, an SBST approach

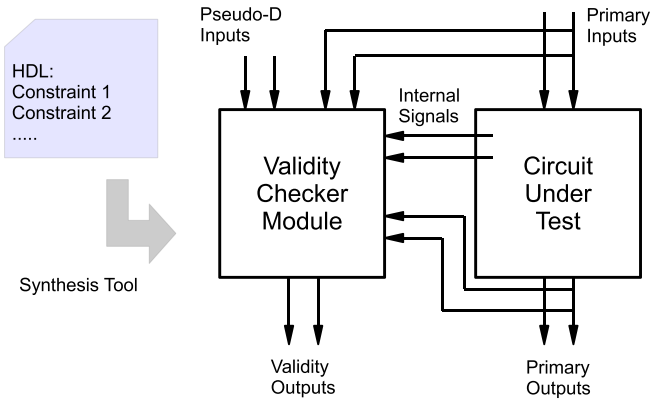

Fig. 3. Validity checker module (VCM).

comprises the following steps. First, a test program and corresponding data are uploaded to the memory accessible by the microprocessor under test. Then, the program is executed and the final memory content is downloaded. Finally, the memory content is compared with a golden version, i.e., a fault-free execution. Consequently, only valid instructions and data words can be applied to the data bus. In addition, hardware interrupts cannot be controlled by software means. Furthermore, a fault effect can only be observed if written in the memory at the end of the test. The described constraints are exemplary for a realistic SBST environment. However, the concrete constraints can differ depending on the requirements imposed by the environment. Therefore, an approach is required, which can flexibly model these requirements and constrain the generated TSs accordingly.

We propose a so-called validity checker module (VCM) for the specification of the SBST requirements. The VCM is a circuit that can be specified in a hardware description language (HDL), such as VHDL or Verilog. It is then synthesized to a gate-level netlist and combined with the CUT. The described functional ATPG framework (Section III) is extended to incorporate the constraints specified in the VCM into the TS generation. This will yield TSs, which satisfy all specified constraints. Thus, the VCM serves as an interface for the specification of constraints given by the environment of the CUT. This interface is also very flexible as the addition of new constraints or the modification of existing constraints can be achieved by (re)writing a small amount of the HDL code.

Fig. 3 shows an overview over the VCM concept. In the HDL source file, a test engineer will specify a number of constraints describing the environment of the CUT. A constraint has to be a circuit itself with an arbitrary number of inputs and one output. The output should be 1 if the constraint is satisfied and 0 otherwise and is denoted by validity output. The constraint can be designed as a pure combinational block or may contain storing elements. Its inputs can comprise several types of signals as indicated in Fig. 3. The primary inputs and outputs of the CUT can be utilized as constraint inputs, e.g., for forbidding certain values on these signal lines. An internal signal can serve as an input, e.g., if the validity of the constraint depends on a CUT register value. In addition, we introduce pseudo-D inputs that are not part of the CUT. However, they are required if a constraint, referring to a fault

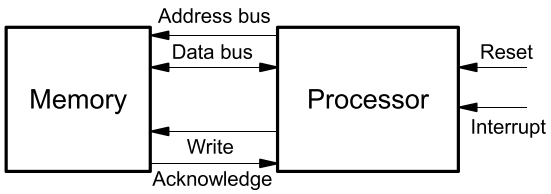

Fig. 4. Typical in-field SBST environment.

effect at a certain signal or bus, has to be specified. For example, if a constraint requires a fault effect to be visible at bus data, then pseudo-D input data_D is introduced, which is handled like a normal circuit input in the VCM. The ATPG framework will then identify data_D as the placeholder for the fault effect at data and connect this line to the corresponding variable in the circuit encoding, which determines whether a fault effect is visible, i.e., a difference between the fault-free and faulty circuit version is visible at data. After all constraints are specified, the VCM can be synthesized to a gate netlist. This netlist is combined with the CUT netlist, as shown in Fig. 3. The resulting netlist is the input for the functional ATPG. The ATPG framework supports the activation and deactivation of each constraint during each step. This is useful for identifying constraints, which invalidated test generation for a specific fault. Furthermore, we distinguish between two types of constraints, namely, invariant constraints and target constraints. An invariant constraint has to be valid in each cycle of a TS, while a target constraint only has to be valid in the last cycle of a sequence.

V. Constraints

In this section, we will describe the constraints that are usually present in a realistic in-field SBST environment and explain how these constraints can be enforced with the VCM (Section IV). We divided the constraints into several subclasses, which outline the challenges related to the generation of an SBST program. First, we will discuss each subclass on an abstract level and then give the concretely implemented constraints for a typical in-field test environment. An overview of the considered SBST environment is shown in Fig. 4. The implemented constraints should be seen as an example and as a basis for the specification of constraints for an arbitrary microprocessor and test environment, since the flexibility of our framework allows a user to easily change, remove, or add constraints in order to model his specific requirements.

A. Hardware-Controlled Inputs

As SBST is solely based on the execution of a software program, it is not possible to access hardware-controlled inputs. This is especially true when considering in-field SBST, where the processor is usually embedded in a larger system. Therefore, the input signals of the processor will be connected to other system modules, such as memory, sensors, or IO controllers, and will not be controllable during SBST. Instead, the SBST program has to work properly without being disturbed by these hardware-controlled inputs. Consequently, the user has to specify HDL constraints, which describe the behavior of these signals during SBST execution. For example, an external interrupt may be constantly inactive or may only occur within a certain amount of clock cycles. However, describing such behavior on the high abstraction level of an HDL is easily achievable.

In our SBST environment, we consider three hardware-controlled inputs. First, the system reset signal is only allowed to be active once at circuit initialization and then has to remain inactive. This is reasonable as activating the signal at certain clock cycles would be hard to control in an in-field SBST scenario. Second, analogous to the previous constraint, hardware interrupts are not allowed and have to remain inactive. Third, we assume a memory with a response time of one clock cycle. Therefore, we require the memory acknowledgment signal to be always active. If a memory with another response time or interface is used, constraints corresponding to this memory protocol have to be implemented at this point.

B. Valid Memory Content

In an in-field SBST scenario, we are only able to control the program and data memory content. Depending on the actual processor under test, this content has to be further restricted. In most cases, the program memory is supposed to solely contain instructions from the processor instruction set. This requires the user to implement a constraint, which comprises the encoding of all instructions.

In our test environment, we applied the following constraint. If the processor is loading an instruction, only valid instructions are allowed to be applied at the data bus. For this purpose, the encoding of all instructions from the processor instruction set has to be contained in the implementation of this constraint. If a data word is loaded, no restrictions are applied.

C. Program Memory Coherence

As the objective is to generate a program, it has to be possible to map a computed TS to memory. This requires the coherence of the program memory, i.e., fetching an instruction from one memory address should always return the same instruction. A naive approach could store all accessed instructions in the VCM, i.e., implement a program memory. However, this would induce a large search space and would not be feasible. Hence, we implement this requirement by preventing operations, which decrement the program counter. This can be achieved by restricting branch instructions accordingly and forbidding system functions and exceptions. As this approach is very restrictive and may decrease the fault coverage for certain modules, test macros can be utilized. The idea of a test macro is to specify a small state machine, which requires a certain sequence of instructions. For example, this can be used to allow backward branches, which are usually required for testing branch prediction units.

At the power-ON, the processor is supposed to set the program counter to a well-defined value. The constraint is enforced by forbidding jump instructions to arbitrary memory locations and system functions. Furthermore, branch instructions are only allowed to increase the program counter by a fixed value. This value is chosen by taking into account the number of pipeline stages, which take place after loading the

branch instruction until its execution. The described constraint will, therefore, enforce a monotonic increase in the program counter and avoid several fetch operations from the same memory address. However, this restrictive implementation may prevent testing some parts of the processor. In order to cope with this challenge, the constraint is extended by two macros, which are described in the following.

Increasing the program counter by a fixed value results in high instruction addresses being hard to reach. Therefore, the first macro enables reaching these addresses more easily. It consists of two jump and link instructions (jump to target address and store return address) and one store instruction. The first jump can target an arbitrary memory address above a user-defined threshold; the second jump will then return to the memory address after the first jump. Finally, a store instruction for the return address will be performed. In case of a fault in the address logic, this control flow will be manipulated and the store instruction will not be performed or store a wrong value in the memory.

A branch prediction module will usually require the execution of a branch instruction at a certain memory address several times in order to be activated [10]. Therefore, the second macro enables backward branches in a restricted way. The macro enforces a conditional branch instruction, whose condition is initially false and, therefore, the branch is not taken. Then, the condition is set to true, and a second branch instruction returns to the first conditional branch instruction. As its condition is now true, it will branch to a store instruction, which finishes the macro. A fault in the branch prediction module will cause a branch to a wrong memory address and, therefore, the store instruction will not be executed. Please note that the performance faults [17] in the branch prediction unit, i.e., faults that only cause a wrong branch prediction, cannot be detected due to the fault detection condition, which states that only the memory content after program execution is observable. However, performance faults would require cycle-accurate monitoring of the circuit outputs. Consequently, performance faults are untestable in an in-field SBST scenario.

Finally, the constraint requires that either only one of the macros is active or both are deactivated. In the latter case, branches are restricted to a fixed value. The functional ATPG will then choose one of these three possibilities to test a given fault.

D. Data Memory Coherence

In addition to the program memory coherence, the data memory coherence also has to be guaranteed. This means that two consecutive load instructions accessing the same memory address have to return the same data word. If a store instruction has been executed on a memory address, then all following load instructions accessing the same address have to return the previously stored data word. This requirement can be implemented by defining a starting address for the data memory and by then performing all memory load and store instructions on consecutive memory cells.

The data memory coherence constraint is implemented by initially loading a predefined value into a general purpose register r. The loaded value determines the starting address of the data memory. The value of r serves as the target address for all load and store instructions. After one of these instructions has been executed, the value of r is incremented. No other manipulations of r are allowed. This constraint will, therefore, cause all load and store instructions to target distinct memory cells in the data memory.

E. Fault Detection

In an in-field SBST scenario, it is only possible to observe the memory content after the execution of the test program. Therefore, a fault effect has to be persistently stored in the memory in order to be observed. For this purpose, the pseudo_D inputs (Section IV) can be utilized. The constraint that enforces the fault detection has to require a memory store operation, while a pseudo_D input (related to a memory-controlling output) is active, i.e., a fault effect affecting the store operation is visible. For example, requiring the pseudo_D input of the data bus to be active, while a store is performed, will cause storing faulty data.

For our test environment, the implemented constraint requires that either the memory write signal is faulty or the memory write signal is active and a fault effect is visible at either the address or the data bus. The first requirement will lead to a store instruction, which is only executed in either the fault-free or the faulty case. The latter requirement will cause a store instruction, which either writes to an erroneous memory address or writes an erroneous data value into the memory.

F. SBST Program Extraction

Generating a TS for a fault with the described constraints will return a sequence of patterns consisting of valid instructions and data words. By simulating the sequence, we can determine the memory address, which is outputted by the processor at the address bus for fetching an instruction or loading a data word. The implemented constraints will enforce that each instruction and data word is using a unique address in the memory. The SBST program can be extracted by mapping each instruction to its corresponding memory address. Furthermore, the data words required by the load instructions can be preloaded into the corresponding memory cells. Therefore, a memory mapping is generated, which realizes an SBST program together with its required data.

G. Application to Other Fault Models

The approach detailed in this paper considers stuck-at faults. However, this approach can be extended to other fault models. In [26], a functional ATPG algorithm for small-delay faults was proposed, which is based on the same solver as the functional ATPG in this paper. Riefert *et al.* [22] combined the small-delay fault ATPG with the VCM described in Section IV.

VI. Experimental Results

An MIPS-like processor [27] was used to prove the viability and the effectiveness of the proposed approach. This processor comprises a five-stage pipeline, data forwarding, and

TABLE I
EXPERIMENTAL RESULTS (PERCENTAGE WITH REGARD TO COMPLETE FAULT LIST)

Module	No constraints				Combinational constraints				All constraints			
	untest	abort	FC	FE	untest	abort	FC	FE	untest	abort	FC	FE
pf	0.19	0	99.81	100	0.42	0	99.58	100	0.42	1.52	98.06	98.48
ei	0	0.34	99.66	99.66	0.47	1.36	98.17	98.64	0.47	0.47	99.06	99.53
di	0.05	1.18	98.77	98.82	1.35	1.53	97.12	98.47	2.40	1.90	95.70	98.10
ex	3.32	0.31	96.37	99.69	3.52	0.24	96.24	99.76	4.42	0.68	94.90	99.32
mem	14.13	2.53	83.34	97.47	14.57	10.04	75.39	89.96	14.91	9.55	75.54	90.45
renvoi	5.29	0.19	94.52	99.81	5.29	0.45	94.26	99.55	5.31	0.11	94.58	99.89
banc	0	0	100	100	0.16	0	99.84	100	0.16	0	99.84	100
syscop	19.62	0	80.38	100	23.12	0.58	76.29	99.42	23.23	0.03	76.74	99.97
bus_ctrl	0.17	2.59	97.24	97.41	1.25	10.07	88.68	89.93	2.72	18.50	78.78	81.50
predict	0.04	1.62	98.34	98.38	0.11	0.14	99.75	99.86	0.11	4.84	95.05	95.16
total	2.40	0.56	97.04	99.44	2.85	0.69	96.46	99.31	3.14	1.84	95.02	98.16

branch prediction. We utilized the available register transfer-level description.[3] The VHDL code was synthesized with Synopsys Design Vision using an in-house developed library. The resulting gate netlist contained 18 279 gates and 1966 flip-flops that were all considered for stuck-at test generation. The uncollapsed fault list contains 111 024 faults and is collapsed to 54 181 faults. The VCM containing all described constraints was specified in VHDL and comprises ∼400 lines of code, which resulted in a synthesized netlist consisting of 1387 gates and 55 flip-flops. All experiments were executed on one core of an Intel Xeon processor running at 3.3 GHz and being equipped with 64 GB of RAM. The timeout for each call to the utilized CIP solver was 60 s. One solver call required no more than 3 GB.

A. Evaluation of Contraints

We executed three runs with differing sets of applied constraints. First, we executed an experiment with no constraints applied (no constraints). This corresponds to an end-of-manufacturing test scenario. Second, only the constraints described in Sections V-A and V-B were considered. These constraints are implemented as small combinational blocks and are, therefore, denoted by combinational constraints. Third, the remaining constraints from Sections V-C to V-E, which correspond to more complex sequential blocks, are additionally considered. This experiment is denoted by all constraints. The first run required 32 h, the second run required 29 h, and the third run required 65 h. The runtime decrease in the second run, in comparison to the first run, is due to the combinational constraints, which restrict the search space in a suitable way, and thus simplify the task of the solver. The third run requires significantly more time as the additional constraints, especially the fault detection condition, invalidate several simple solutions, which were valid in the first two runs. While the runtime is relevant, the reader should note that alternative approaches are much more costly as they require a skilled engineer with in-depth processor knowledge to manually generate an SBST program instead of computer runtime.

[3]The available version contains a bug, which always executes a conditional branch, even when the condition does not hold. As the correct execution of a conditional branch instruction is required for the execution of a program, we fixed this bug.

Table I lists the results of the three runs. We give detailed results for each of the processor modules. The subdivision corresponds to the modules given in the available VHDL code. The columns untest and abort give the percentage of faults, which are untestable in the considered scenario, and are aborted, respectively. The column FC gives the fault coverage of the generated TSs. The column FE gives the fault efficiency, i.e., the percentage of faults that can be either tested or proved as being untestable. Compared with other approaches utilizing BMC [12], we have a low abort ratio as our approach can identify all structurally untestable faults and a significant amount of functionally untestable faults. Consequently, aborts are avoided, while processing these faults. The TSs generated with no constraints have a total length of 15 389 clock cycles. Considering the combinational constraints, it yields TSs with a total length of 12 601 clock cycles. When applying all constraints, 17 162 clock cycles are required.

B. Evaluation of Proposed Optimizations

The propagation sequence cache (Section III-E) has been applied to all runs. In order to assess the effectiveness of the cache, we executed an additional evaluation run with no constraints applied and without utilizing the propagation sequence cache. In this evaluation run, we processed all 1588 TSs that were generated by the original run with the cache. For each of these sequences, the evaluation run tried to generate one sequence starting from the same starting state and targeting the same fault as the corresponding sequence from the original run. The described evaluation approach ensures that the original run with cache and the comparison run without cache will always process the same problem instances and, therefore, provide comparable runtimes. The original run utilizing the cache required 72 002 s for generating the 1588 TSs and extracting the propagation sequences. The comparison run without utilizing the cache required 100 837 s for processing the corresponding 1588 problem instances. Out of these instances, 17 were aborted, i.e., no pattern could be generated without utilizing the cache. Thus, the application of the propagation sequence cache could improve the TS generation runtime by ∼28%. This demonstrates that the knowledge gained from already generated TSs can be automatically extracted and used beneficially for the

further ATPG process. Out of the 1588 generated sequences, 1068 propagation sequence cache hits occurred. For all of these cache hits, the cached sequence provided a valid fault propagation sequence, i.e., it was not corrupted by the targeted fault. This shows that assuming the propagation of a latched fault effect to be mostly fault independent (Section III-E) is reasonable.

We also evaluated the effectiveness of the proposed heuristic for the prediction of aborts (Section III-F) by executing an additional experimental run with combinational constraints and without the described heuristic. The results show that the proposed heuristic reduces the runtime by 26% and additionally misclassifies only 0.13% of the total number of faults as aborts.

C. Comparison With Other Approaches

In [28], an approach for the generation of an SBST program was proposed, which is based on the manual development of test programs for the different modules of a processor. Gizopoulos *et al.* [28] evaluated the processor, which is also considered in this paper. They are able to achieve fault coverage of 95.08%. It has to be noted that they consider an older version of the processor, which does not contain a branch prediction unit. However, it can be concluded that our automatic approach achieves the comparable fault coverage (95.02%) without the need for manual effort and in-depth processor knowledge. In addition, it can classify 3.14% faults as untestable and achieve a fault efficiency of 98.16%. The manual generation of test algorithms is particularly tedious and time expensive when considering pipeline structures with bypassing and data forwarding. Furthermore, certain faults can become untestable due to the constraints imposed by a realistic SBST scenario. Due to standards and regulations for safety-critical systems, it is also important to identify these faults. By activating or deactivating certain constraints in the VCM, our approach allows not only to prove untestability of a fault but also to identify the constraint, which made the fault untestable.

In [18] and [19], also the same processor, as considered in this paper, is evaluated and the results are given. However, the processor is synthesized using differing libraries. Furthermore, the fault coverage given in both works is computed by considering only structurally testable faults, i.e., structurally untestable faults are collapsed. Thus, the fault coverage given here is rather comparable to the fault efficiency of our approach. Furthermore, Lu *et al.* [18] utilize so-called micro observations which means that the fault effect can be observed at each clock cycle. This makes it rather comparable to our run, which considers only combinational constraints. In [18], 98.46% fault coverage can be achieved, while our approach considering combinational constraints achieves 99.31% fault efficiency. Compared with the fault coverage of 97.31% achieved by [19], we achieve a fault efficiency of 98.16%. As detailed, these numbers cannot be compared exactly but still indicate the effectiveness of our approach.

For comparison, we also evaluated the effectiveness of random TSs. Fig. 5 shows the evolution of the fault coverage

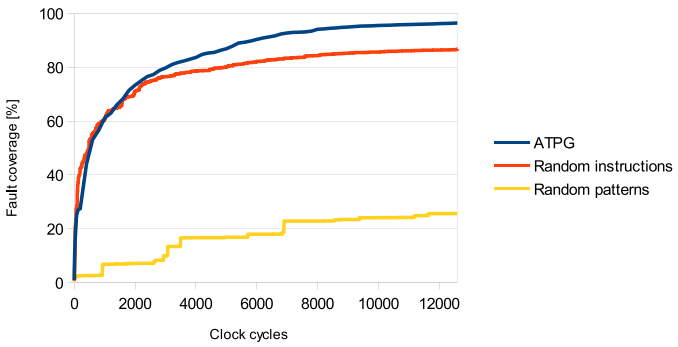

Fig. 5. Comparison of fault coverage saturation with random patterns.

over the number of TS clock cycles for different test sets. The upmost curve (ATPG) is based on the test set generated by our functional ATPG with combinational constraints applied and saturates at 96.46%. The second curve (random instructions) is based on a test set where the reset and interrupt signals are always inactive; the memory acknowledgment signal is always active and random instructions are applied. This corresponds to the combinational constraints. This curve saturates at about 86.66%. In the beginning, this curve rises more steeply than the fault coverage of the ATPG-generated test set as the fault list, processed by the ATPG framework, is sorted modulewise. Consequently, the ATPG will process all faults from one module before considering the next module, while random instruction will sooner address easy-to-detect faults from all modules. The lowest curve (random patterns) is based on completely random patterns. This test set is only able to reach ∼25.63% fault coverage. Fig. 5 shows that constrained random patterns can be used to achieve basic fault coverage. However, they are not able to achieve high fault coverage as they fail to test hard-to-detect faults, which require a very specific TS.

VII. Conclusion

In this paper, we have presented a framework, which allows for the first time to automatically generate an effective in-field test program for a pipelined processor. We have detailed the basis of this framework, which is a functional ATPG engine capable of efficiently generating a TS for a fault or proving its untestability. Several optimizations of the functional ATPG engine have been developed. Furthermore, the flexible interface of our framework has been described, which allows the specification of arbitrary constraints, induced by an in-field test environment, on an abstract level. We have listed typical constraints, which exist in an in-field processor test environment and illustrated their integration into our framework. Finally, extensive experimental results have been reported, which show the effectiveness of the approach and give significantly improved runtimes with regard to the previous works. In particular, the experimental results show that our fully automatic test program generation method can achieve the same stuck-at fault coverage as manual approaches. Furthermore, it can take into account any constraint stemming from an in-field SBST environment, and identify untestable faults.

REFERENCES

[1] M. Psarakis, D. Gizopoulos, E. Sanchez, and M. Sonza Reorda, "Microprocessor software-based self-testing," *IEEE Design Test Comput.*, vol. 27, no. 3, pp. 4–19, May/Jun. 2010.

[2] S. M. Thatte and J. A. Abraham, "Test generation for microprocessors," *IEEE Trans. Comput.*, vol. 29, no. 6, pp. 429–441, Jun. 1980.

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

[4] L. Chen, S. Ravi, A. Raghunathan, and S. Dey, "A scalable software-based self-test methodology for programmable processors," in *Proc. Design Autom. Conf.*, Jun. 2003, pp. 548–553.

[5] F. Corno, E. Sanchez, M. S. Reorda, and G. Squillero, "Automatic test program generation: A case study," *IEEE Design Test Comput.*, vol. 21, no. 2, pp. 102–109, Mar. 2004.

[6] P. Parvathala, K. Maneparambil, and W. Lindsay, "FRITS—A microprocessor functional BIST method," in *Proc. IEEE Int. Test Conf.*, Oct. 2002, pp. 590–598.

[7] I. Bayraktaroglu, J. Hunt, and D. Watkins, "Cache resident functional microprocessor testing: Avoiding high speed IO issues," in *Proc. IEEE Int. Test Conf.*, Oct. 2006, pp. 1–7.

[8] S. Gurumurthy, M. Pratapgarhwala, C. Gilgan, and J. Rearick, "Comparing the effectiveness of cache-resident tests against cycleaccurate deterministic functional patterns," in *Proc. IEEE Int. Test Conf. (ITC)*, Oct. 2014, pp. 1–8.

[9] S. Di Carlo, P. Prinetto, and A. Savino, "Software-based self-test of set-associative cache memories," *IEEE Trans. Comput.*, vol. 60, no. 7, pp. 1030–1044, Jul. 2011.

[10] E. Sanchez and M. S. Reorda, "On the functional test of branch prediction units," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 9, pp. 1675–1688, Sep. 2015.

[11] G. Theodorou, S. Chatzopoulos, N. Kranitis, A. Paschalis, and D. Gizopoulos, "A software-based self-test methodology for on-line testing of data TLBs," in *Proc. IEEE Eur. Test Symp. (ETS)*, May 2012, p. 1.

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

[13] L. Lingappan and N. K. Jha, "Satisfiability-based automatic test program generation and design for testability for microprocessors," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 15, no. 5, pp. 518–530, May 2007.

[14] Y. Zhang, A. Rezine, P. Eles, and Z. Peng, "Automatic test program generation for out-of-order superscalar processors," in *Proc. IEEE 21st Asian Test Symp.*, Nov. 2012, pp. 338–343.

[15] V. Singh, M. Inoue, K. K. Saluja, and H. Fujiwara, "Instruction-based self-testing of delay faults in pipelined processors," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 11, pp. 1203–1215, Nov. 2006.

[16] S. Gurumurthy, R. Vemu, J. A. Abraham, and D. G. Saab, "Automatic generation of instructions to robustly test delay defects in processors," in *Proc. 12th IEEE Eur. Test Symp.*, May 2007, pp. 173–178.

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

[18] T.-H. Lu, C.-H. Chen, and K.-J. Lee, "Effective hybrid test program development for software-based self-testing of pipeline processor cores," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 19, no. 3, pp. 516–520, Mar. 2011.

[19] Y. Zhang, H. Li, and X. Li, "Automatic test program generation using executing-trace-based constraint extraction for embedded processors," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 21, no. 7, pp. 1220–1233, Jul. 2013.

[20] P. Bernardi *et al.*, "On-line software-based self-test of the address calculation unit in RISC processors," in *Proc. 17th IEEE Eur. Test Symp. (ETS)*, May 2012, pp. 1–6.

[21] P. Bernardi, M. Bonazza, E. Sanchez, M. S. Reorda, and O. Ballan, "On-line functionally untestable fault identification in embedded processor cores," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Mar. 2013, pp. 1462–1467.

[22] A. Riefert, L. Ciganda, M. Sauer, P. Bernardi, M. S. Reorda, and B. Becker, "An effective approach to automatic functional processor test generation for small-delay faults," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, Mar. 2014, pp. 1–6.

[23] A. Riefert, R. Cantoro, M. Sauer, M. S. Reorda, and B. Becker, "On the automatic generation of SBST test programs for in-field test," in *Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE)*, 2015, pp. 1186–1191.

[24] S. Kupferschmid, M. Lewis, T. Schubert, and B. Becker, "Incremental preprocessing methods for use in BMC," *Formal Methods Syst. Design*, vol. 39, no. 2, pp. 1–20, 2011.

[25] S. Reimer, M. Sauer, T. Schubert, and B. Becker, "Incremental encoding and solving of cardinality constraints," in *Proc. 12th Int. Symp. Autom. Technol. Verification Anal.*, 2014, pp. 297–313.

[26] 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 *Proc. IEEE Int. Test Conf. (ITC)*, Nov. 2012, pp. 1–8.

[27] *miniMIPS*, accessed on Apr. 16, 2015. [Online]. Available: http://opencores.org/project,minimips

[28] D. Gizopoulos *et al.*, "Systematic software-based self-test for pipelined processors," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 11, pp. 1441–1453, Nov. 2008.

Andreas Riefert (M'12) received the master's degree in computer science from the University of Freiburg, Freiburg im Breisgau, Germany, in 2012, where he is currently pursuing the Ph.D. degree with the Department of Computer Science, and the Computer Architecture Group headed by B. Becker.

His current research interests include test and reliability of microprocessors.

Riccardo Cantoro (M'13) received the M.Sc. degree in computer engineering from the Politecnico di Torino, Turin, Italy, in 2013, where he is currently pursuing the Ph.D. degree with the Department of Computer Engineering.

His current research interests include microprocessor testing.

Matthias Sauer (M'10) received the master's and Ph.D. degrees from the University of Freiburg, Freiburg im Breisgau, Germany, in 2010 and 2013, respectively.

He is currently a Postdoctoral Research Associate with the Department of Computer Science, University of Freiburg, where he is also with the Computer Architecture Group headed by B. Becker. His current research interests include the application of formal methods to hardware security and the timing analysis of digital circuits.

Matteo Sonza Reorda (F'16) received the M.S. degree in electronics and the Ph.D. degree in computer engineering from the Politecnico di Torino, Turin, Italy, in 1986 and 1990, respectively.

He is currently a Full Professor with the Department of Control and Computer Engineering, Politecnico di Torino. His current research interests include test of systems-on-chip and fault-tolerant electronic system design.

Bernd Becker (F'86) was an Associate Professor of Complexity Theory and Efficient Algorithms with J. W. Goethe-University, Frankfurt, Germany. He joined the University of Freiburg, Freiburg im Breisgau, Germany, in 1995, where he is currently a Full Professor with the Faculty of Engineering. A focus of his research is the development and analysis of efficient data structures and algorithms in VLSI computer-aided design (CAD). The development of symbolic methods for test and verification of digital circuits and their integration in the industrial flow is one of the major achievements of his work. More recently, he has been involved in verification methods for embedded systems and test techniques for nanoelectronic circuitry. He has authored over 300 papers in peer-reviewed conferences and journals and has been on the program and organizing committees of numerous major international conferences. His current research interests include CAD and test and verification of (digital) circuits and systems (VLSI CAD).

Prof. Becker is a member of Academia Europaea. He acts as the Co-Speaker of the DFG Transregional Collaborative Research Center Automatic Analysis and Verification of Complex Systems with project partners from the University of Freiburg, the University of Saarland, Saarbrücken, Germany, the University of Oldenburg, Oldenburg, Germany, and the Max Planck Institute of Computer Science, Saarbrücken.