Automated Software-Based In-field Self-Test Program Synthesis

Artjom Jasnetski, Raimund Ubar, and Anton Tsertov

***Abstract*—This paper presents a methodology to automate functional Software-Based Self-Test program development. We rely on the previously published research on modeling processors using subclass of acyclic directed graphs called High-Level Decision Diagrams (HLDD). The HLDD model of the processor gets generated from its Instruction Set Architecture. The HLDD model is then used together with beforehand prepared assembly program templates in the generation of the complete self-test program. The research presented in this paper includes examples of test generation for the 32-bit SPARCv8 microprocessor Leon 3. The experimental results demonstrate that automatically generated SBST program obtains comparable to the state-of the art fault coverage data.**

***Index Terms*—microprocessor, software-based self-test (SBST), automatic test program generation, high-level decision diagrams (HLDD) synthesis.**

I. Introduction

ADVANCES of modern technology in manufacturing and design of microprocessors are continuously increasing the difficulty of digital circuit test [1]. Therefore, testing of constantly scaling complex digital systems like microprocessors, has been a challenge for decades. Software-based self-test method has emerged, and became a very promising competitor to the widely used, but slow, intruisive structural test [2] [3], and effective, but very expensive functional test [1]. The core idea of SBST approach is to use the resources of microprocessor to test itself, by running test programs. The nature of this method implies such features as nonintruisiveness, low cost and compatibility with at-speed and in-field testing [4]. This method was accepted by industry [5], and is complementing functional and structural methods within manufacturing process.

Furthermore, interest for this method was growing in frames of in-field test for processor-centric systems in safety-critical applications. Since, functional and structural test methods are not suitable for in-field test, SBST becomes very attractive solution [6] [7]. The academia was motivated to put more effort into studying SBST for in-field test, after publication of IEC 61508 for industrial safety systems, ISO 26262 for automotive applications, and release of DO0254.

Concurrently, big interest is gathered around the automation of SBST approach, since the complexity of manual test program generation can be inexcusably high. Automated SBST [7]–[9] can reduce the test development cost, and thereafter price of a product.

SBST approaches can be divided into two major groups - structural and functional. Structural approaches, such as [10]–[15], are based on test generation using information from lower level of design (gate- or RTL-level description) of processor under test. Functional, in its turn, is using instruction set architecture (ISA) information of the processor under test. Since in most cases structural information of commercial products is intellectual property held under NDA, the solution based on functional SBST is exclusive for in-system or in-filed test.

One of the first methods among functional SBST proved to be efficient, was proposed by Shen And Abraham [16]. Framework Vertis, capable of pseudo-random test sequence generation based on ISA information, has been proposed. Similar solution - FRITS (Functional Random Instruction Testing at Speed) [17] is based on test program generation on random instruction sequences with pseudo-random data. It suits well for wafer test, due to it's cache-resident nature. Alternative cache-resident method for production testing [5] using random generation mechanism proves, that high cost functional testers can be replaced by this SBST approach, without significant loss in fault coverage. Alternative approach, based on so-called evolutionary algorithm, was proposed by Corno et al. [18]. Test program is being composed of the most effective code snippets (in a question of SAF coverage), which were distinguished by constant reevaluation. Due to it's reevaluation-centric nature, this method is not capable of in-field test generation, due to lack of structural information. Later research concentrates on test approaches for specific processor parts like pipeline, branch prediction mechanism [19] or caches [20]. Gizopoulos et al. in [21] are proposing a method, which can enhance SBST program in order to bring more coverage to pipeline logic and also memory addressing (12% for miniMIPS and OpenRISC1200 processors). Further approach for testing the pipeline was made by Bernardi et al. [22]. The proposed strategy involves the activation of faults related to the data hazards and register forwarding logic in processor core, and later research concentrates on decode stage of the pipeline [6].

Nevertheless, none of the state-of-the-art methods have so far tried to develop well formalized high-level (e.g. behavioral level) fault models for coping with hard-to-test faults and fault masking problems at higher levels with ultimate goal to improve test quality and to achieve compact test programs. Additionally, coverage of wide specter of fault classes is left unmeasured, due to lack of methods for simulation and lack of theoretical basis for identification.

A. Jasnetski, R. Ubar and A. Tsertov are with Department of Computer Engineering, Tallinn University of Technology, Tallinn, Estonia (e-mails: ajasn@ati.ttu.ee, raiub@ati.ttu.ee, anton.tsertov@ttu.ee)

This work was jointly supported by IT Academy of Estonia, EU through European Regional Development Fund, H2020-ICT-2014-1 644905 project IMMORTAL as well as by the institutional research funding IUT 19-1 of the Estonian Ministry of Education and Research.

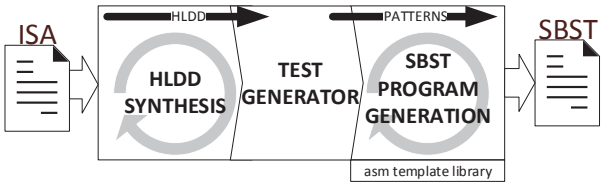

Fig. 1. General concept of SBST generation tool

Our previous work in SBST field is focused on adapting methodology of High-Level Decision Diagrams (HLDD) for modeling of microprocessors and faults [23]. In paper [23], we introduced a new concept for generating tests for microprocessors. The proposed approach considers program generation using MUTs modeling at behavioral level derived from ISA. HLDD model sets are synthesized from the ISA, hence, considered as a behavioral level model. The instruction list of MP is converted into a network of HLDD graphs where each graph represents a sub-circuit. Hierarchical approach is used for test generation: the control functions are tested exhaustively (by conformity test), but the data operands for testing the data path are generated by traditional gate-level ATPG (scanning test). Our latest work [24] adds new fault modeling idea for high-level faults, where experimental results obtained by formal generation of test program for the Parwan [25], [26] microprocessor are presented.

In this paper we develop our previous concepts into the tool, which automatically generates test programs for microprocessors. For that purpose we have proposed an algorithm of instruction set analysis. Generalizing the known method of BDD synthesis based on Shannon expansion of Boolean functions allows high level expansion of predicate expressions. The general concept of the tool is shown in Figure 1. The Tool consists of three modules: HLDD synthesizer, test vector generator, and SBST generator - synthesizer for converting test vectors into test-programs using beforehand prepared test code templates. The capabilities of the tool are demonstrated on two microprocessors - on the Parwan 8-bit microprocessor, and on Leon 3 32-bit microprocessor.

The paper is organized as follows: Section II presents the basis of HLDD synthesis procedure. Section III is demonstrating the HLDD synthesis functionality of proposed tool on example of Leon 3 microprocessor. Section IV is dedicated to fault modeling using HLDD diagrams. Section V gives an overview of test program generation functionality, and experimental results. The results, published in this paper are confirming the applicability of approaches presented in previous works [23] [24].

II. High-Level Decision Diagram Synthesis

In [27], High-Level Decision Diagrams (HLDD) were introduced, and a method was proposed for synthesis of HLDDs from Data Flow Diagrams (DFD). As the first step of synthesis, the DFD was transformed by symbolic execution into a Structural Table of Automaton (STA) [28]. In case of microprocessors when their behavior is given by the instruction set, such STA can be generated directly without symbolic execution of the model.

Consider an example of a fragment of a digital system is shown in Table I.

TABLE I
Description of an automation

Instruction	Control constraints			Data assignment statements
I_k	x_1	x_2	x_3	
I_1	1	3	0	$y_1 = F_1(X)$
I_2	1	2		$y_1 = F_2(X)$, $y_2 = F_3(X)$
I_3	3		2	$y_1 = F_4(X)$
I_4		4	1	$y_2 = F_5(X)$
- - -				

Each row in Table I represents a state transfer in the automaton. In case of the microprocessor represented at high-level by its set of instructions, we can represent by each row the functionality of an instruction as follows.

The left-most cell in each row denotes the name (or the number) of the instruction I_k, the cells of the subtable Control constraints represent the code of the the related instruction word I_k split into the codes (x_1, x_2, x_3) of different subfields of the instruction format, and the cells of the subtable Data assignment statement represent the functional activities $y_k = F_k(X_k)$ of the related instruction where y_k denotes the output functional variable of the related functional block (e.g. the output register of ALU), and X_k represents the data variables involved as arguments in the data manipulation operation F_k.

For all the left-hand side variables y_k in Table I we create HLDDs which will describe the behavior of these variables during execution of the related instructions. We assume that the variables y_1 and y_2 represent the results of functions $F(X)$.

Consider the data in Table 1 as a set of tuples $N = \{N_k\}, N_k = (C_k, S_k)$ where C_i is a set of logical constraints given for the instruction I_k in the subtable control constraints, and S_k is a set of assignment statements. Each statement $s \in S_k$ (denoted by shortly by $y_{k,s} = F_{k,s}$) is an algebraic expression, which will be fulfilled if the set of constraints C_i is satisfied. By collecting all the statements s from N for a left-hand variable y we can represent the behavior of y as

$$y = \bigvee C_i F_{i,s}, \tag{1}$$

where the constraints C_i represent conjunctions of predicates weighted by the respective expression $F_{i,s}$.

As an example, according to the formula 1 the Table I can be represented now as the two predicate formulas:

$$y_1 = (x_1 = 1)(x_2 = 3)(x_3 = 0)F_1(X) \vee$$

$$(x_1 = 1)(x_2 = 2)F_2(X) \vee (x_1 = 3)(x_3 = 2)F_4(X)$$

$$y_2 = (x_1 = 1)(x_2 = 2)F_3(X) \vee (x_2 = 4)(x_3 = 1)F_5(X)$$

From these predicate formulas, the HLDDs for the variables y_1 and y_2 can be derived in a similar way as BDDs are derived by using Shannon factorization [29] for Boolean functions. The only difference is that instead of Boolean factorization we will use multi-valued factorization, depending on the possible number of values of the constraint variables x_k. The HLDDs created by factorization for y_1 and y_2 are depicted in Fig 2.

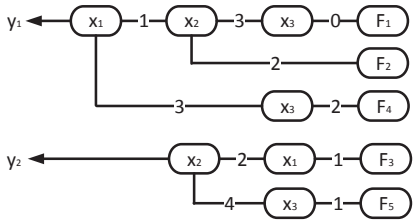

Fig. 2. HLDDs created by factorization of formulas for y_1 and y_2

Using the model of HLDDs we can simulate the instructions by tracing the graphs according to the values of the instruction variables (x_1, x_2, x_3). If a terminal node will be reached then the value of the graph variable y is updated by calculation of the value of the expression in the related terminal node. If no terminal node will be reached then the value of graph variable y will not be updated.

III. ISA Based HLDD Synthesizer

The methodology for generation of High-Level decision diagrams from instruction set architecture presented in previous section, is used by the proposed tool. The instruction set information is primary input data for HLDD synthesis. In order to process ISA automatically, it should be represented in a machine readable way. Usually the information about ISA is formatted and composed differently, making the universal parsing process nearly impossible.

First, we suggest to bring the ISA description to common ground. Thus, in this paper we outline two formats (CSV, XML) to generalize the description of Leon 3 ISA. In case of CSV format, each instruction field name, width and value must be provided using such syntax: $\%name\% = \%width\%'b\%value\%$. In case of XML format, each field must have own tag, where name, width and value are provided as follows: $< \%name\% = \%width\%'b\%value\% >< \%name\% >$. The proposed HLDD Synthesizer tool is capable to read both formats: CSV table and XML.

As a case study, microprocessor Leon 3 was used. This processor was chosen to represent the complexity of modern processor cores, which is suitable to show the scalability of proposed approach. Table II shows the complexity difference between two processors used in case study of this paper. An integer unit module of Leon 3, as a core component, is used as an unit under test.

TABLE II
PARWAN and Leon 3 IU comparison

	PARWAN	LEON 3 Integer Unit
Bit depth	8	32
Instructions	16	46
Architecture	Custom	SPARCv8
Gates	1480	15161

Leon 3 is a microprocessor with SPARCv8 architecture [30]. SPARCv8 instructions can be divided into four groups: memory, control, ALU and miscellaneous. Each instruction is aligned to a specific format. SPARCv8 architecture has 6 different formats for instruction set listed in Table III.

TABLE III
SPARCv8 instruction formats

1	op	disp30					
2	op	rd		op	imm22		
3	op	a	cond	op2	disp22		
4	op	rd		op3	rs1	i=0	asi	rs2
5	op	rd		op3	rs1	i=1	imm13	
6	op	rd		op3	rs1	opf		rs2

The fields from instruction format table can be divided in two types - operational fields (*op, op2, op3, i, a, asi, cond, opf*) and register related fields (*rs1, rs2, rd, imm*). This information must be handled differently, in order to build a proper HLDD diagram. Operational code fields must be marked differently from register fields.

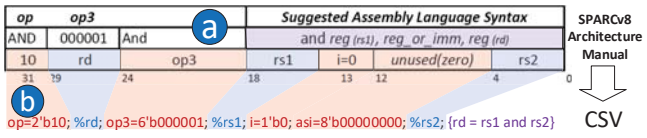

Fig. 3. CSV and XML representation of AND instruction

An example of *AND* instruction of SPARCv8 architecture is provided in Figure 3. This figure represents the process of modifying the instruction description from architecture reference manual into machine readable format. In the given example *AND* instruction description (Figure 3.a) is cut from SPARCv8 architecture manual. By it's format *AND* instruction belongs to the ALU instruction group. Additional information about instruction fields is also taken from manual - instruction word consists of four operational fields - *op, op3, i* and *asi*, and three register fields - *rd, rs1* and *rs2* (Table III, line 4). Based on this information, an entry containing instruction field names and their length, can be added to the CSV file (Figure 3.b). *op, op3, i* and *asi* fields have constant values, but *rd, rs1* and *rs2* fields are dynamic, since holding information about register index. Dynamic nature of register index is represented with symbol %. Complementary information should be provided as parameters separately, in order to sort dynamic fields by their specification - if it is a source register, destination register or immediate value.

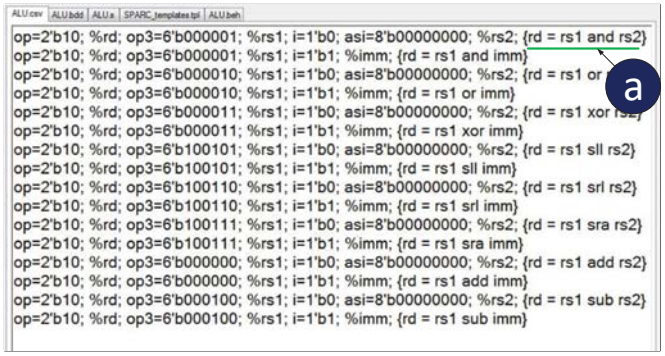

Fig. 4. Part of SPARCv8 ALU instructions

In case of *AND* instruction, *rd* is a destination register index and *rs1, rs2* are source register indexes. Additionally, CSV entry should contain the description of operation (Figure 4.a),

for the *AND* operation in hand is *rd = rs1 and rs2*. Complementary, but important information about microprocessor architecture should be also added as separate parameters. Such is the data about register amount and their width, needed at the stages of HLDD generation and test synthesis.

Correctly composed CSV with complementary files is holding needed data to build HLDD diagram, representing the behavior of the system (or it's part) under test. As an example, a small subset of SPARCv8 instruction set, representing ALU instructions, is shown in Figure 4. The HLDD graph, synthesized from SPARCv8 ALU-type instruction list is shown in Figure 5. In case of *AND* instruction, the destination register, represented by instruction field *rd*, becomes the output of the graph. Then, path of consequent nodes from output to the leaf of graph is build from operational fields of *AND* instruction. Functional description of *AND* instruction becomes the leaf. Such way of modeling allows to store the behavior of the system as follows: the result of operation *rs1 and rs2* will be stored to *rd*, if operational instructions fields *op, op3, i* and *asi* are holding specific values. As a consequence, each node represents an element of the control part, and leafs represent data path of the modeled system.

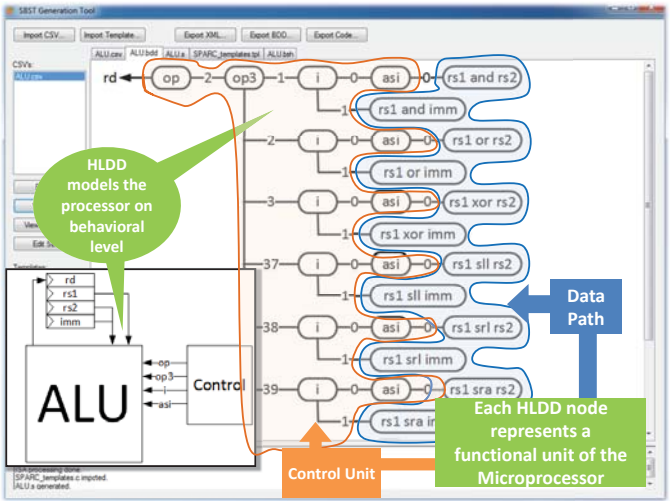

Fig. 5. HLDD graph synthesised from SPARCv8 instructions

IV. ISA Based High-Level Fault Modeling

In the ISA based HLDD model, the nodes of the decision diagram (DD) are classified into two groups: internal nodes, and terminal nodes. Internal nodes represent the control functions of the system, and terminal nodes represent the data path functions of the system.

The instruction words of microprocessors are usually split into several fields. This corresponds to partitioning of the instruction variable into concatenation of the field variables e.g. as *I = OP.A1.A2*, where *I* is the instruction variable, *OP* is the operation code, whereas *A1*, and *A2* denote register addresses of the first and second operands, respectively. In the HLDD model, to each of these field variables related internal nodes correspond. On the other hand, each of these nodes represent sub-circuits which are responsible for addressing the operands and controlling the operation related to the value of *OP*. The nodes *OP*, *A1*, and *A2* represent a path in the HLDD, which will be activated if the instruction *I* is called. The path terminates in the terminal node of the HLDD labelled by a functional expression to be processed in the data path of the system if the instruction *I* is called.

Each path in a DD describes the behavior of the system in a specific mode of operation. The faults having effect on this behavior are associated with nodes along the activated path. In case of a control fault, the path activated by instruction *I*, will be corrupted, and the effect of the fault will cause incorrect leaving the path in the faulty node. In this case, a wrong terminal node will be reached instead of the terminal node which should have been reached at instruction *I*. In case of the data fault, the functional expression in the terminal node of the activated path will be corrupted.

As the *fault universe* to represent all possible control malfunctions in a system, we assume any corruption in the behavior of non-terminal nodes in HLDDs, expressed in the following ways [31]:

1) output edge of a node is broken;
2) output edge of a node is constantly activated;
3) instead of the activated edge, another edge or a set of edges are simultaneously erroneously activated.

The practical meaning of this fault universe stands in application of the idea to test exhaustively the behavior of each non-terminal node at all possible values of the node variable.

The fault universe of control faults can be expanded with the universe of data path faults related to terminal nodes of HLDDs to get the full fault universe of the system under test. To do this, in each terminal node, and for each expected value v of the functional expression of the node, we introduce a dummy output edge into a new dummy terminal node labelled with the value v as a constant. In such an extended HLDD, all the node related faults can be activated in a uniform way.

Several optimizations of the described exhaustive fault universe can be undertaken, based on either transforming exhaustive test into pseudo-exhaustive one, or transforming the functional test into the structural one.

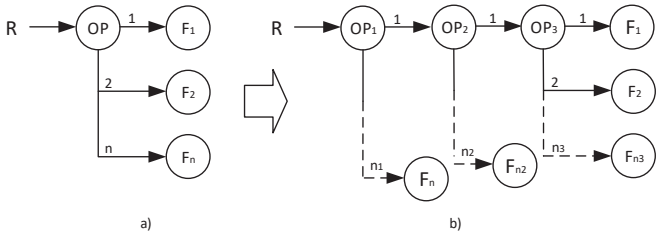

Fig. 6. Transformation of HLDDs to reduce the complexity of model

In Figure 6, it is shown how the exhaustive test concept can be substituted by the pseudo-exhaustive approach to reduce the control fault universe of the system. Two possible HLDDs are presented for representing the behavior of the same subsystem with output register *R*. The subsystem is controlled by a set of *n* instructions. The instruction variable *I* can be represented as a concatenation of the field variables in two ways: *I = OP.A1.A2*, or *I = OP1.OP2.OP3.A1.A2*, resulting in two different HLDDs.

Instead of testing exhaustively the node *OP* in the HLDD (Figure 6a) for all *n* values of the instruction subfield variable *OP*, we may test separately and exhaustively the component variables *OP1*, *OP2*, and *OP3* in the functionally equivalent HLDD (Figure 6b), which corresponds to traditional pseudo-exhaustive test [31]. The complexity of the test generation task, and the length of the resulting test can be using the HLDD in Figure 6b considerably reduced due to the reduction of the number of output edges of internal nodes in the graph, compared to the HLDD in Figure 6a:

$$n_1 + n_2 + n_3 << n = n_1 \times n_2 \times n_3.$$

On the other hand, the exhaustive test of the functions *Fk* in terminal nodes, can be replaced either by also pseudo-exhaustive tests [32], or by structurally generated test patterns using any traditional low-level ATPG (if the related implementation details of the system are given). In the latter case, a combined use of hierarchical test generation may be used, exploiting both, ISA-based HLDDs, and structurally synthesized BDDs [33].

To organize test generation for each HLDD node, using HLDD-based fault universe, two steps are to be processed: activation of the node under test, and sensitizing the faults of the activated node.

To activate internal node *m* under test needs assignment of proper values to the node variables in the HLDD, so that the following paths were activated: (2) a path from the root node to *m*, and (3) a subset of non-overlapping paths from all output edges of the node *m* to a subset of terminal nodes $M^T(m)$. To activate a terminal node *m* under test needs to activate only a single path from the root node to *m*. Sensitizing the faults of the activated internal node m needs to solve the following equations as constraints when testing the node *m* [2]:

$$\forall m^T \in M^T(m) : [f(m^T) \neq \Omega], \tag{2}$$

$$\forall m_i, m_j \in M^T(m), i \neq j : \forall k[f_k(m_i) < (f_k(m_i) * f_k(m_j))] \tag{3}$$

where Ω = ZERO (or ONE), and the symbol * stands for logic OR (or logic AND), depending on the technology implemented in MP [34], [35]. Here, ZERO denotes a binary vector (000), and, similarly, ONE stands for (111). The index *k* refers to the bit number of the data words. Satisfaction of the constraints (2) and (3) guarantees that the expected and erroneous test responses will be distinguished at any corruption of the activated HLDD node *m* under test due to a fault in the sub-system, represented by the node.

The described high-level fault model defined for HLDDs, together with the node activation concept, can be regarded as a generalization of the classical gate-level stuck-at fault (SAF) model for high-level representations of digital systems. Both represent a *node based fault model* in decision diagrams [33]. The only differences are in the number of output edges of the nodes, and in the number of terminal nodes in the decision diagrams. In both cases, the node variables are tested exhaustively: two test patterns are needed for the Boolean variables labelling the nodes in BDDs whereas the number of patterns needed for HLDD nodes is equal to the number of output edges of the node.

The described concept of satisfaction of the constraints (2) and (3) is similar to the extended conditional SAF model [36]–[38] developed for Boolean level test generation of physical defects inside complex gates in digital circuits. In the latter case, additional conditions map the impact of defects into SAF at related BDD nodes, whereas in case of the ISA based HLDD model, the constraints (2) and (3) specify the reasons of the corruptions in behaviors of nodes under test.

V. SBST Generation with HLDD Model

The targets of test generation for a microprocessor using the HLDD model are not the instructions each of them taken as a whole as in traditional cases. Instead of that, the targets are smaller functional entities represented by the nodes of HLDDs. The terminal nodes represent selected data path functional entities (sub-circuits of ALU), and the nonterminal nodes represent the selected control functional entities related to the subfields of instruction words. Since the HLDD nodes as test targets represent smaller functional units than the instructions as a whole, it makes possible to use pseudo-exhaustive testing of the processor control part and to cope in this way better with the complexity of the test problem. Instead of full exhaustive testing of all operation codes we test (pseudo)exhaustively its independent parts, guided by the HLDD internal nodes. For testing terminal nodes we use test data generated for ALU at the gate level.

From above, two approaches of testing, different for terminal and nonterminal nodes, result: conformity test for the control part (internal HLDD nodes), and scanning test for data path (terminal HLDD nodes) [39].

A. Conformity tests

The test program is synthesized on the high-level directly from the HLDD model, and the data for the test program are generated to satisfy the constraints (2) and (3).

Algorithm 1. Conformity test for the control part (test for a nonterminal node m).

1) Control data (instruction code) generation: activate in the HLDD a path lm from the root node to the node m under test, and for each output k of the node m a path l_k to a terminal node m_k^T with operation $f(m_k^T)$. The value of $z(m)$, which represents a sub-field of the instruction code, will be cyclically varied during the pseudo-exhaustive test execution.
2) Data path initialization: find the proper sets of data values $D(m)$ which satisfy the constraints (2) and (3).
3) Test implementation: the generated instruction should be repeated for all the values $z(m)$ of the node m under test, updated dynamically by these values, and using always the same data operands in $D(m)$.

B. Scanning tests

The test program is synthesized on the high-level directly from the HLDD model, and the data for the test program are

generated by a traditional gate-level ATPG using the given descriptions of the data blocks.

Algorithm 2. Scanning test program generation for testing the data path (terminal node m) for operation $f(m)$.

1) High-level test generation: activate in the HLDD a path l_m from the root to the terminal node m.
2) Low-level test generation: find the proper sets of data values $D(m)$ for the arguments in $f(m)$.
3) Test implementation: the generated instruction should be repeated for all the values of $D(m)$, i.e. for all of the arguments of $f(m)$.

C. SBST generation with a tool

The proposed tool, is utilizing these ideas in the test program generation process. The result of test generation, is a test pattern, which holds encoded information about instruction and operands (Figure 7(A)). Since, there is normally no framework available to handle test program for microprocessor in machine code, the task of SBST generator (see Figure 1) is to decode patterns (Figure 7(B)) obtained from test generator into assembly instructions. This is done by using predefined templates stored in the assembly code library. As a result, the test program, compiled from code templates is made. It can be edited further, in order to improve the fault coverage, or add code parts, which can not be generated automatically.

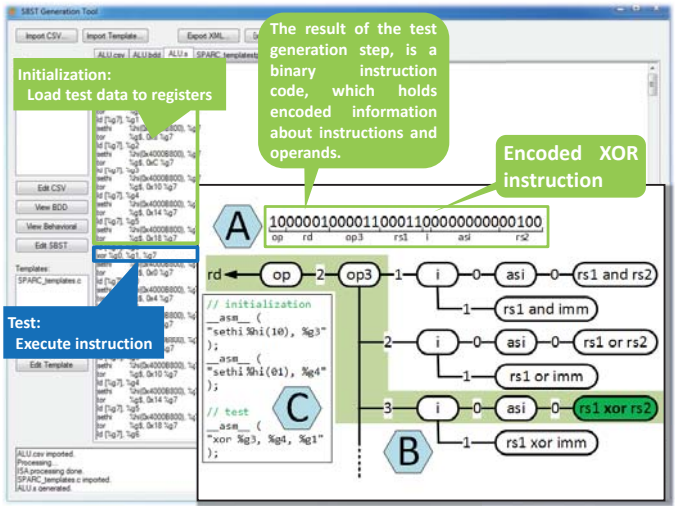

Fig. 7. Example of test program generation

The test program generation process is shown in general in Figure 7. The process can be divided into two parts - initialization and test. The initialization part is loading test data into registers, and the test part is combining the instruction fields from the library into the full instruction code. In Figure 7(C), a subset of generated test program is shown. The first part of code represents an initialization process. The registers are filled with data (partially shown). Every register is loaded with data before testing each instruction, so that to avoid fault masking [24]. Then, the test part is being generated. In the section A of Figure 7, a test pattern string, retrieved from the test generator is shown. Since the instruction fields are known (Table III), the test generator can walk trough HLDD nodes, and construct the corresponding instruction. A path, highlighted in green leads to the instruction XOR (Figure 7(B)). Test generator is looking for XOR assembly code template in premade SPARCv8 library, and modifies it to read data from registers, specified by the test generator (rd, $rs1$ and $rs2$ fields) (Figure 7(A, C)).

Test program generation is strongly affected by the modeling level made in previous steps. The more details can be extracted from instruction set architecture, the better test program can be generated. Specific behavior of the processor can be hidden or even invisible from the ISA point of view, and simple list of instructions not enough to cover the realistic structural results.

The exact fault coverage can be calculated by gate-level fault simulation. The not detected gate-level faults may belong to the class of redundant faults. Otherwise, to detect these faults, low-level ATPGs can be used for generating additional test operands.

The main contribution of the proposed method of SBST generation is to substitute existing labour extensive low-level test generation methods with fast high-level test generation, accompanied with as well fast low-level fault simulation to obtain the exact evaluation of the test quality.

VI. Experimental Results

In the experiments, a test program was generated automatically for the Integer Unit (IU) of Leon 3. The Fault simulation of the generated test program was made using TetraMAX [40] software. The fault simulation framework is described in details in our previous work [24]. The results are shown in Table IV. "HLDD test program" shows a result of fault simulation with automatically generated test program. "HLDD test program random" is automatically generated, but test operands are selected randomly. "Leon3 startup test" is a test program supplied with processor description files [41], which tests memory and peripherals on startup. "TetraMAX ATPG" represent a local fault coverage of patterns generated by a sequential ATPG tool. However not all generated test vectors are functionally correct (they cannot be reproduced during normal CPU operation), hence the coverage is overestimated. The row "HLDDtp + Leon 3 st" represents the fault coverage result for both test programs. According to calculated fault coverage, we can assume, that those programs can complement each other and cover additional faults in components of Integer Unit. From the point of view of test engineer, our tool is a good opportunity to improve fault coverage (about 5%), with minimal effort.

The low fault coverage is explained by the fact that not all instructions using the Integer Unit were taken into account for building the HLDD model. Extension of the model for the full instruction list needs further research.

However, to demonstrate the feasibility and efficiency of the approach for the case when the HLDD model covers full set of microprocessor instructions, the experimental research was carried out for the PARWAN microprocessor. The results in comparison with previous research on PARWAN [42], [43] are depicted in Table V, which demonstrates the superiority of our results. The PARWAN was used because of the availability of published results for that microprocessor.

TABLE IV
LEON 3 INTEGER UNIT FAULT SIMULATION RESULTS

Leon 3 Integer Unit	Faults total/testable	FC, %	Fault simulation, in minutes
HLDD test program	42780 / 38847	43,84	34
HLDD test program random		41,40	78
Leon 3 startup test		40,93	22
HLDD tp + Leon 3 st		45,26	78
Tetra max ATPG		72,89	2496 *

** time used for ATPG and fault simulation together*

TABLE V
PARWAN FAULT SIMULATION RESULTS

Module	#Faults	Fault coverage %		
		Proposed method	[43]	[42]
AC	156	**99.3**	99.30	99.30
IR	228	**99.4**	96.40	98.60
PC	590	**99.3**	99.00	89.20
MAR	342	**99.2**	96.40	97.20
SR	130	**99.0**	96.80	98.90
ALU	956	**99.3**	98.00	98.50
SHU	310	**100**	99.20	94.10
Control	648	**89.8**	84.40	88.30
Total	2960	**98.04**	96.19	95.51

In this experiment with Leon processor we concentrated on test generation for the data path of the IU, which is related to fetch, decode and memory stages of the pipeline [30], other stages are tested indirectly. Moreover, since we concentrated our efforts only for IU-related instruction groups (ALU and memory), then a lot of control part functionality was not covered by HLDD test, like state, flags, traps, FPU and Coprocessor instructions and controling rotation of register windows (exclusive for SPARC architecture). This explains the low fault coverage. The further work will be to extend the not yet covered hardware part responsible for the unused instruction groups. Still, the composed program was able to discover a considerable amount of faults, which weren't covered by default Leon 3 test program. Test data, which was selected by test generator gives better fault coverage, than random data. The fault simulation time is increased in case of random data, because the amount of data is multiple times more than in case of deterministic test data generation.

VII. CONCLUSION

In this paper we developed first, a novel algorithm and a tool for formal synthesis of the HLDD model for a given set of the instructions of the microprocessor under test, and second, a tool for automated software-based self-test program generation for microprocessors based on the HLDD model.

The novelty of described tool is an automation of test program generation. The capabilities of the tools are demonstrated on the 8-bit microprocessor PARWAN, and on the 32-bit Leon 3 SPARCv8 microprocessor. In combination with the fault simulation tools, the described in the paper test generation tool promises to be a helpful instrument for test engineers. The positive fault coverage results, obtained during evaluation of the test program with TetraMAX simulator, are confirming the feasibility of the proposed approach.

However, because of the unique features of the investigated Leon microprocessor architecture, a fully-automatic approach for its full instruction set is not available at the moment. Still, test engineer can modify generated test program in order to increase fault coverage.

A field of the future work exists. A backwards analysis of assembly program, can show, which paths are covered on the HLDD model of the microprocessor, and make an approximate fault coverage estimation of the test program. This will decrease test program development time, since fault simulation, especially sequential, takes lots of time and computational resources.

REFERENCES

[1] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer Publishing Company, Incorporated, 2013.

[2] E. B. Eichelberger and T. W. Williams, "A logic design structure for lsi testability," in *Papers on Twenty-five Years of Electronic Design Automation*, ser. 25 years of DAC. New York, NY, USA: ACM, 1988, pp. 358–364. [Online]. Available: http://doi.acm.org/10.1145/62882.62924

[3] E. B. Eichelberger and T.W.Williams, "A logic design structure for LSI testability," in *Proc. of the DAC*, 1977, pp. 462 – 468.

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

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

[6] P. Bernardi, R. Cantoro, L. Ciganda, E. Sanchez, M. S. Reorda, S. D. Luca, R. Meregalli, and A. Sansonetti, "On the in-field functional testing of decode units in pipelined risc processors," in *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Oct 2014, pp. 299–304.

[7] A. Riefert, R. Cantoro, M. Sauer, M. S. Reorda, and B. Becker, "A flexible framework for the automatic generation of sbst programs," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 10, pp. 3055–3066, Oct 2016.

[8] 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 *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2014, pp. 1–6.

[9] M. Schlzel, T. Koal, S. Rder, and H. T. Vierhaus, "Towards an automatic generation of diagnostic in-field sbst for processor components," in *2013 14th Latin American Test Workshop - LATW*, April 2013, pp. 1–6.

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

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

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

[13] C. H. P. Wen, L.-C. Wang, and K.-T. Cheng, "Simulation-based functional test generation for embedded processors," *IEEE Transactions on Computers*, vol. 55, no. 11, pp. 1335–1343, Nov 2006.

[14] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-based self-testing of embedded processors," *IEEE Transactions on Computers*, vol. 54, no. 4, pp. 461–475, April 2005.

[15] C. H. Chen, C. K. Wei, T. H. Lu, and H. W. Gao, "Software-based self-testing with multiple-level abstractions for soft processor cores," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 5, pp. 505–517, May 2007.

[16] J. Shen and J. A. Abraham, "Native mode functional test generation for processors with applications to self test and design validation," in *Proceedings International Test Conference 1998 (IEEE Cat. No.98CH36270)*, Oct 1998, pp. 990–999.

[17] P. Parvathala, K. Maneparambil, and W. Lindsay, "Frits - a microprocessor functional bist method," in *Proceedings. International Test Conference*, 2002, pp. 590–598.

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

[19] E. Sanchez and M. S. Reorda, "On the functional test of branch prediction units," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 9, pp. 1675–1688, Sept 2015.
[20] S. D. Carlo, P. Prinetto, and A. Savino, "Software-based self-test of set-associative cache memories," *IEEE Transactions on Computers*, vol. 60, no. 7, pp. 1030–1044, July 2011.
[21] D. Gizopoulos, M. Psarakis, M. Hatzimihail, M. Maniatakos, A. Paschalis, A. Raghunathan, and S. Ravi, "Systematic software-based self-test for pipelined processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 11, pp. 1441–1453, Nov 2008.
[22] P. Bernardi, R. Cantoro, L. Ciganda, B. Du, E. Sanchez, M. S. Reorda, M. Grosso, and O. Ballan, "On the functional test of the register forwarding and pipeline interlocking unit in pipelined processors," in *2013 14th International Workshop on Microprocessor Test and Verification*, Dec 2013, pp. 52–57.
[23] R. Ubar, A. Tsertov, A. Jasnetski, and M. Brik, "Software-based self-test generation for microprocessors with high-level decision diagrams," in *Proc. of the Latin-American Test Workshop*, 2014, pp. 1–6.
[24] A. Jasnetski, J. Raik, A. Tsertov, and R. Ubar, "New fault models and self-test generation for microprocessors using high-level decision diagrams," in *Proc. of the International Symposium on Design and Diagnostics of Electronic Circuits Systems*, 2015, pp. 251–254.
[25] Z.Navabi, *Analysis and Modeling of Digital Systems*. McGraw-Hill, 1993.
[26] [Online]. Available: http://mesdat.ucsd.edu/ lichen/260c/parwan/
[27] R.Ubar, J.Raik, A.Karputkin, and M.Tombak, "Synthesis of high-level decision diagrams for functional test pattern generation," in *Proc. of 16th Int. Conference MIXDES*, Lodz, Poland, Jun. 2009, pp. 519–524.
[28] S. Baranov, *Logic and System Design of Digital Systems*. Tallinn, Estonia, year =: TUT Press.
[29] B. R. Drechsler, *Binary Decision Diagrams*. Boston, MA, USA: Kluwer Academic Publishers, Boston, 1998.
[30] S. international Inc. The sparc architecture manual, version 8. [Online]. Available: http://www.gaisler.com/doc/sparcv8.pdf
[31] E. J. McCluskey, "Verification testinga pseudoexhaustive test technique," vol. Vol. C-33, pp. 541 – 546, 07 1984.
[32] A. S. Oyeniran, A. Jasnetski, A. Tsertov, and R. Ubar, "High-level test data generation for software-based self-test in microprocessors," in *2017 6th Mediterranean Conference on Embedded Computing (MECO)*, June 2017, pp. 1–6.
[33] R. Ubar, "Test synthesis with alternative graphs," in *IEEE Design and Test of Computers*, 1996, pp. 48 – 59.
[34] S. M. Thatte and J. A. Abraham, "Test generation for microprocessors," vol. C-29, no. 6, June 1980, pp. 429–441.
[35] D. Brahme and J. A. Abraham, "Functional testing of microprocessors," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 475–485, June 1984.
[36] R. D. Blanton and J. P. Hayes, "Properties of the input pattern fault model," in *Proceedings International Conference on Computer Design VLSI in Computers and Processors*, Oct 1997, pp. 372–380.
[37] S. Holst and H. J. Wunderlich, "Adaptive debug and diagnosis without fault dictionaries," in *2008 13th European Test Symposium*, May 2008, pp. 199–204.
[38] F. Hapke, W. Redemund, A. Glowatz, J. Rajski, M. Reese, M. Hustava, M. Keim, J. Schloeffel, and A. Fast, "Cell-aware test," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 9, pp. 1396–1409, Sept 2014.
[39] R.Ubar, J.Raik, and H.-T.Vierhaus, *Design and Test Technology for Dependable Systems-on-Chip*. IGI Global, 2011.
[40] [Online]. Available: http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Test/Pages/TetraMAXATPG.aspx
[41] [Online]. Available: http://www.gaisler.com/products/grlib/grlib-gpl-1.4.1-b4156.tar.gz
[42] L. Chen and S. Dey, "Software-based self-testing methodology for processor cores," in *IEEE Trans. on CAD of IC and Systems*, vol. 20, no. 3, March 2001, pp. 369 – 380.
[43] Y. Zhang, H. Li, and X. Li, "Software-based self-testing of processors using expanded instructions," in *Proc. of 19th IEEE Asian Test Symposium*, 2010, pp. 415 – 420.

Artjom Jasnetski received his M.Sc.degree in computer engineering from Tallinn University of Technology, Estonia in 2013 and currently he is Ph.D. student at Tallinn University of Technology. His research interests include such topics as microprocessor test, digital system modelling, ISP and HW driver implementation.

Raimund Ubar is a member of IEEE, professor at Tallinn University of Technology, and Head of the Centre for Integrated Electronic Systems and Biomedical Engineering in Estonia. He received PhD degree in 1971 from the Bauman Technical University in Moscow, and DSc degree in 1987 from the Latvian Academy of Sciences. His scientific interests include computer science, design for testability and diagnostics of technical systems. Raimund is a member of European Test Technology Technical Committee, a member of Estonian Academy of Sciences, and Golden Core member of IEEE Computer Society. He was awarded from the Estonian Government by White Cross Orden of III Class, and by Meritorious Service Award of the IEEE Computer Society.

Anton Tsertov received his M.Sc. and Ph.D. degrees in computer engineering from Tallinn University of Technology, Estonia in 2007 and 2012 respectively and currently holds the position of researcher in Tallinn University of Technology. His research interests include such topics as system and board level test, high-level system modelling, microprocessor functional and structural test.