Automatic Generation of On-Line Test Programs through a Cooperation Scheme

L. Ciganda[1], M. Gaudesi[1], E. Lutton[2], E. Sanchez[1], G. Squillero[1], A. Tonda[3]

[1]Politecnico di Torino – Italy

[2]Université Paris-Sud – France

[3]Institut des Systèmes Complexes – France

***Abstract*— Test programs for Software-based Self-Test (SBST) can be exploited during the mission phase of microprocessor-based systems to periodically assess hardware integrity. However, several additional constraints must be imposed due to the coexistence of test programs with the mission application. This paper proposes a method for the generation of SBST on-line test programs for embedded RISC processors, systems where the impact of on-line constraints is significant. The proposed strategy exploits an evolutionary optimizer that is able to create a complete test set of programs relying on a new cooperative scheme. Experimental results showed high fault coverage values on two different modules of a MIPS-like processor core. These two case studies demonstrate the effectiveness of the technique and the low human effort required for its implementation.**

Keywords- SoC, pipelined processors, on-line testing, software-based self-test, Group Evolution

1. INTRODUCTION

The increasing use of electronic components based on microprocessors for safety and mission-critical applications drives manufacturers to integrate more and more dependability features in their products.

Aiming at increasing product dependability, as well as coping with market and standard requirements, producers must include a series of auditing processes during the lifecycle of the product and in-mission periodic testing campaigns for error detection. For example, car manufacturers are adopting the ISO/DIS 26262 [1] standard that demands the inclusion of the on-line self-test techniques as essential test processes in critical electronic vehicle parts to insure high quality and mission safety throughout the product useful life.

One of the most interesting techniques to achieve these results is based on the application of on-line test programs. Usually, a set of test programs is first devised to test the whole processor or some modules inside of it, and then periodically run during the device lifecycle. It is possible to define a test program as a carefully devised piece of code aiming at testing some part of the processor core resorting only to the processor instruction set and exploiting the available resources of the processor. Interestingly, the test program is not only in charge of exciting processor faults, but also contributes to bringing results to an observable point. Even though this technique was firstly introduced more than 20 years ago [2], it easily matches with on-line testing requirements and it is therefore gaining again the attention of the research community.

In Software-Based Self-Test (SBST) the application of test programs is carried out by the very same processor core on the electronic device. The technique can be easily applied to on-line testing performing periodically few simple steps:

- The mission application is interrupted;
- The test program is uploaded in the code memory;
- The test program is executed by the processor core;
- Results are gathered and saved;
- The mission application restarts.

Test programs must follow all the requirements and constraints established for the device under testing. Indeed, transforming an SBST test set to an on-line SBST test set may not be possible, or it may require excessive effort. For example, as described in [9] [15], on-line tests must *preserve processor status*, *have a reduced execution time*, and *preserve memory integrity*. Several works presented in the literature show why software-based techniques are often preferable to hardware approaches. However, while SBST generation approaches are described in [3] [4] [5] [6], there is not a mature technique able to cope with all the constraint introduced by the current on-line testing requirements.

In this paper we concentrate on the generation of SBST programs oriented to microprocessor on-line testing. The proposed strategy exploits a new evolutionary concept that automatically generates a whole set of test programs, taking advantage of their complementarities. The proposed approach achieves very good coverage figures, and requires lower generation time and memory footprint when compared with analogous strategies. The final results gathered on a freely available version of a pipelined microprocessor clearly cope with on-line constraints requiring low human intervention.

The rest of the paper is organized as follows: section 2 recalls some important concepts that support the rest of the paper; section 3 introduces the proposed approach, describing in details the novelties of the proposed strategy. Sections 4 and 5 present the case study and outline the results gathered on a pipelined processor core. Finally, the last section concludes the paper, and drafts the future works.

1550-4093/12 $26.00 © 2012 IEEE
DOI 10.1109/MTV.2012.17

2. BACKGROUND

SBST is an emerging alternative for identifying faults during normal operation of the product, by performing on-line testing [7]. Several reasons push this choice: SBST does not require external test equipment, it has very little intrusiveness into system design and it minimizes power consumption and hardware cost with respect to other on-line techniques based on circuit redundancy. It also allows at-speed testing, a feature required to deal with some defects prompted by deep submicron technology advent.

Evolutionary algorithms (EAs) have been little, but steadily, used in the CAD community during the past 20 years [8]. EAs provide an effective methodology for trying random modifications, where no preconceived idea about the optimal solution is required. Being based on a population of candidate solutions, they are more robust than pure hill climbing. Both small and large modifications of existing solutions are possible, but with different probabilities. Recombination makes it possible to merge useful characteristics from different solutions, exploring efficiently the search space. Furthermore, EAs are quite simple to set up, require no human intervention when running and are inherently parallel. Finally, it's easy to trade-off between computational resources and quality of the results.

A versatile toolkit, named μGP (MicroGP), has been developed at Politecnico di Torino in the early 2000s and it is now available under the GNU Public License [10]. It must be noted that μGP does not aim at creating a program to solve generic problems, but rather to *optimize* realistic assembly-language programs for very specific purposes.

The last release of μGP is composed of three clearly separated blocks: *the evolutionary core* that is able to generate syntactically correct assembly programs using the information about the structure of the language from the *constraint library*; every new generated program is then evaluated resorting to an *external evaluator* that provides the evolutionary core with a measure of its goodness or usefulness (usually obtained by simulation or fault simulation). Borrowing the term from biology, this value is called *fitness*, and is used during the rest of the generation, together with some structural information, for enhance and improve the test programs. At the end, this process provides the user with the best test program.

The simplest, and probably most common, approach used for finding solutions of complex compound problems with an EA is based on iterative runs. That is, since a single solution is not able to solve the whole complex problem, the EA is set to solve *as much as possible* of the problem. Then, the solved sub-problems are removed from the original problem, and a new run of the EA is used to solve the remaining part. Iteration after iteration, the EA provides a set of solutions that is cumulatively able to solve the whole problem. However, the process is plainly inefficient, and the final set is likely to be sub-optimal and oversized. Moreover, a deep knowledge of the problem is required to identify the solved portions.

For instance, if the problem is attaining 100% fault coverage on a circuit, then at each step detected faults are removed from the fault list (the compound problem), and in each iteration the evolutionary test program generator is run against the undetected faults. This approach can lead to good solutions, as [11], despite the fact that test programs in the final test set are likely to cover multiple times the same faults.

The approach proposed in this paper is based on a recent development in the evolutionary computation field, sometimes called *group evolution* or *group selection*. The EA is asked to determine a set of solutions, cumulatively able to solve the whole problem in an optimal way. Thus, only a single iteration is required, leading to a better solution and a significant reduction of computational time [13].

3. CONCURRENT SBST GENERATION OF TEST PROGRAMS FOR ON-LINE TESTING

The final goal of this paper is to introduce an effective strategy to generate SBST programs, suitable to be periodically run during the device mission. It is important to notice that the proposed approach uses the fault coverage obtained by the test programs against the stuck-at faults of the module under consideration as fitness function, that is, as feedback value,.

In a preliminary step, the user is required to select one by one every processor module requiring a test program. Then, targeting the selected module, the user is required to follow the proposed approach based on three steps, namely: *ISA distillation*; *concurrent test set generation*; and *final on-line set up*. At the end of the whole process, the proposed approach produces a complete test set to be applied during mission time.

The proposed method requires low human intervention, as described in the following steps:

1. ISA distillation

This step requires the intervention of the user, by wisely selecting among the processor instructions belonging to the processor ISA, a reduced set of instructions able to firstly excite and then propagate the results of the elaboration to a well-defined memory location.

The instructions gathered during this step are placed into μGP's constraints library. In addition, the user must also set μGP to limit test programs' size, in order to cope with on-line testing constraints.

This step can be performed using two different strategies:

a) Single instructions constraint

The user collects and elaborates the constraints library for μGP by gathering most of the instructions able to interact with the module under consideration. The user must be able to gather different types of instructions that prepare and apply the input values to the module under consideration, and also observe the obtained results. However, the user is not required to organize the gathered instructions in a particular way; instead, the whole organizing process is left to μGP.

b) Atomic block-based constraint

The user, supported on his/her own experience, devises a block of instructions, called atomic block, that must be as general as possible, so as to cope with different module requirements.

Roughly speaking, the atomic block targets the module under consideration by applying suitable values to the module inputs using controlling instructions; then, the test program executes a target instruction that excites the module; and finally, results are propagated to the processor outputs through

instructions that, for example, save results in appropriate memory locations.

Figure 1 shows the structure of a sample atomic block oriented to test the arithmetic module in a pipelined processor.

```
1.    rA ← RNDM (constrained) value
2.    rB ← RNDM (unconstrained) value
3.    rE ← f(rA, rB)
4.    sd_instr rE, M[rA]
```

Figure 1. A sample atomic block pseudo-code.

The first two instructions in Figure 1 (lines 1 and 2) load random values in the registers rA and rB. The value in rA is a constrained random address value (constrained) selected within a writable Test Memory addressing range. On the other hand, the value placed in rB is purely random without any constraint. The target instruction at line 3 manipulates rE by applying the function f(rA, rB). The selected function must be able to excite the module under test. The value in rE is later stored (line 4) in memory completing the observability task of the atomic block for testing the considered module.

It is important to notice that even though the atomic block structure is fixed, in the presented case, µGP is allowed to modify not only the random values but also the involved registers.

Depending on the module under consideration, the user may devise not only one atomic block but a set of them; and then, µGP can be set to freely combine them during the generation phase.

2. Concurrent test set generation

The innovative idea is to generate the complete test set of programs for the module under consideration at the same time, taking into account the constraints for on-line testing. This goal is achieved exploiting *Group Evolution* [12], a new technique able to optimize solutions composed of homogeneous elements. The approach cultivates a population of partial solutions, and exploits non-fixed sets called *groups*. Group Evolution operates on individuals and groups in parallel.

During the evolution, individuals are optimized as in a common EA, and concurrently groups are turned into *teams*. A group in itself does not necessarily constitute a team: teams have members with complementary skills and generate synergy through a coordinated effort which allows each member to maximize its strengths and minimize its weaknesses. A team comprises a group of entities, partial solutions in our case, linked in a common purpose. Teams are especially appropriate for conducting tasks that are high in complexity, whereas leveraging individual cooperation. As a result, the initial sets of random test programs slowly turns into coordinated teams, where each member tackles its specific portion of the problem.

a) Algorithm details

The algorithm proposed is population-based: we generate and keep track of a set of distinct individuals or test programs which share the same structure. In parallel, we manage a set of groups, each one composed by a certain number of individuals. An individual, at any step, is part of at least one group.

The evolutionary algorithm exploits an external evaluator able to assign a fitness value to every individual or test program, as well as to every group considering all individuals composing it.

At the beginning of the evolutionary process, the initial population of test programs is randomly created on the basis of the instructions selected in the previous step (*ISA distillation*). Groups at this stage are randomly determined, and each individual can be part of any number of different groups. Figure 2. shows a sample population with 8 individuals and 4 groups. Notice for example that individual A is part of only one group, while individual B is part of 3 different groups.

The whole evolutionary process follows a generational approach: at each evolutionary step, called *generation*, a certain number of genetic operators is applied to the population. Genetic operators produce new individuals and groups starting from existing individuals or groups.

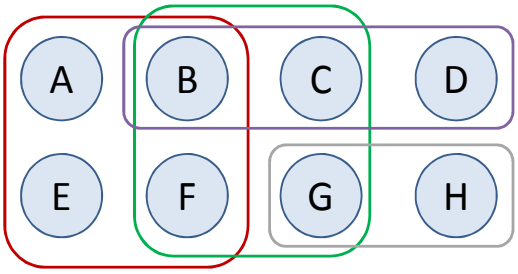

Figure 2. Individuals and Groups in an 8-individual population

The creation phase is made of two different actions at each generation step:

- Application of *group genetic operators*
- Application of *individual genetic operators*

Each time a genetic operator is applied to the population, parents are chosen and offspring is generated. The children are added to the population, while the original parents remain unmodified. Offspring is then evaluated, while it is not compulsory to reconsider the fitness value of the parents again.

Group genetic operators work on the set of groups available in the population. Each operator needs a certain number of groups as parents and produces some offspring groups to be added to the population. The operators implemented are: crossover, union, separation, adding-mutation, and removal-mutation.

Individual genetic operators work as in the classic EA on the population individuals. However, individuals are selected for reproduction considering not only their individual fitness, but also the fitness obtained by the group the individual belongs to. The main novelty, regarding classic evolution, relies in the fact that every time a new individual is generated, for each group the individual is part of, the algorithm generates a new copy of the group including the new individual created by the genetic operator. Since this process could lead to an explosion in the number of groups into the population, the algorithm keeps a new group only if its fitness value is greater than or equal to the one of the parent group.

At the end of every generation step, the group population is ordered and resized according to the number of groups defined in the experiment configuration. If this process leaves some individuals without any reference to a group, those individuals will be removed from the population.

b) Adding-mutation with external support

A new genetic operator is included in the current EA for supporting the inclusion of an individual belonging to the population to a group. This operation may represent a critical task, since the evolutionary algorithm has no other information than the fitness values assigned to the individuals, which may not provide any information regarding the performance the group may achieve with the new individual. Thus, we implement a new genetic operator that receives a group and a series of individuals and then externally evaluates them in order to determine which individual is the most suitable to be added to the selected group.

3. Final on-line set up

At this point, the process provides the user with a series of test programs for the targeted processor modules. However, the final test set may require to be slightly tweaked to actually comply with on-line constraints. Therefore, this step requires once again human intervention, since the testing engineer must verify that the final test set complies with on-line constraints.

The advantages presented by this approach are multiple, with respect to manual approaches, automatic approaches based on classic EA, and random approaches. In particular, as corroborated by the experimental results, the proposed approach:

- Reduces the generation time, since produces at every run a complete group of solutions, instead of a single one;
- Intrinsically thrusts a cooperative effort on the members of a group, reducing the final number of test programs and coverage overlapping;
- Reduces test application time.

4. Case studies and Experimental results

The feasibility and effectiveness of the proposed approach have been evaluated on a freely available processor core called miniMIPS [14]. The processor is synthesized with an in-house developed library, resulting in a circuit containing 33,117 logic gates. The miniMIPS architecture is based on 32-bit buses and includes a 5-stage pipeline.

In the presented work, we concentrate our efforts on two different modules available in the processor: the address calculation module, involved mainly in load and store instructions; and the forwarding unit, used to avoid processor interlocks due to data hazards.

In order to run our experiments, we implement a framework that closes the loop required by the evolutionary optimizer to automatically generate test programs. The framework follows the description given in [15]. Roughly speaking, the whole process is performed as follows: µGP creates a test program that is firstly simulated using the RTL description of the processor core. At the end of this process, the logic simulator gathers the information on the I/O signals of the processor core saving it in a VCD file. Then, the VCD file is passed to a fault simulator that targets only the stuck-at faults of the module under consideration. Then, a coverage value (FC%) is calculated and, finally, it is fed back to the evolutionary optimizer closing the generation loop.

The logic simulation was performed using ModelSim SE 6.4b by Mentor Graphics, while the circuit fault coverage was evaluated using TetraMax v. B-2008.09-SP3 by Synopsys.

1) Address calculation module

The address calculation module is in charge of calculating the addresses for memory access operations performed when *load* and *store* instructions are executed. In pipelined processor cores, this computation is usually performed by a dedicated adder, whose purpose is to sum an offset to a base value to address the RAM memory for reading or writing a value.

Usually, this adder is not part of the processor ALU, and thus it does not perform any arithmetic computations required by instructions like *add* and *sub*. Testing an adder is often deemed as a trivial task, but in the case of the address generation module, controllability and observability are limited, and on-line requirements pose additional constraints.

The criticalities in testing this module with a software-based approach are mainly due to the type of instructions (load and store and all their variants) that activate the address calculation. In fact, a test program including many of such instructions may potentially induce some undesirable effects:

- Store instructions may corrupt the mission data and compromise the correct resuming of the system;
- Load instructions may retrieve data from memory zones (i.e., the parts containing the mission application variables) whose content can hardly be forecasted a priori, therefore compromising the signature generation, no matter how it is calculated.

The adder performing the address calculation that belongs to the synthesized miniMIPS processor is a separated unit within the execution stage, counting 342 logic gates and 1,988 stuck-at faults.

In the following we describe the outcome of the described phases while targeting the address calculation module:

a) ISA distillation

Targeting the address calculation module, we devise a *constraint library* for µGP that exploits a similar atomic block as the one described in [9]; the adopted structure is devised constraining the memory writing and reading spaces in a strictly bounded area that comply with the on-line constraints defined in [9]. Intentionally, we decide not to include additional instructions in the constraint library. The constraint library for the address calculation module is described in a xml file containing 160 lines.

b) Concurrent test set generation

The evolutionary optimizer was set to create an initial population of 30 test programs containing a set of blocks that varies from 1 to 10. The number of groups allowed in the population is 20 and every group may include from 2 to 15 individuals.

In order to support the new genetic operator described in section (3.2.b) a *Perl* script counting 150 lines was implemented. It is important to mention that the external evaluator keeps the fault simulation results of every one of the test programs created by µGP, so that in the case µGP requires

to re-evaluate an individual to assess its performance in a group, the fault simulation process can be skipped.

c) Final on-line setup

In a preliminary run, the test program generation took 432 generations and about 141 hours of computation in a server running Linux OpenSUSE 11.1 OS with 2 CPUs Intel Xeon E5450 @3.00 GHz and 8 GB of RAM, the evolutionary optimizer produces a group composed of 15 individuals that are able to cumulative reach 89.23% fault coverage. In synthesis, every test program belonging to the final best group contains about 6 atomic blocks, and on average the programs contain 35 instructions each. Executing the programs contained in the best group requires about 700 clock cycles. However, each individual test program is run in less than 100 clock cycles.

Let us assume that in order to correctly schedule and run the test programs to execute during the mission cycle, it is required to interrupt the mission application by no more than 75 clock cycles (7.5µs considering a clock frequency of 10 MHz): then, the only modification required on the final test set of programs is to split one of the programs that includes 5 more atomic blocks than the rest of the programs. Thus, the considered program may be divided in two programs in order to do not exceed the clock cycle budget established for every test program.

An additional experiment was performed in order to compare the obtained results. In this case, we exploit an evolutionary approach based on iterative runs [11] creating a final test set that obtained about 87% fault coverage generating 20 test programs whose generation process and general characteristics are similar than the ones used in the previous experiment.

It should be noticed that in the presented experiment using the Group Evolution approach, even though the obtained coverage is slightly lower (less than 2%) than the one obtained in [9], the final test set is most suitable for on-line testing considering the final programs size and execution time. In addition, comparing the proposed approach with [11], the new approach obtained better results also with regards to generation time.

2) Forwarding unit

Register Forwarding (or data bypass) and Pipeline Interlock are functions managed by combinational logic units included in a pipelined microprocessor, to avoid data hazards.

The methods employed to avoid data hazards consist mainly in checking if there is a data dependency between two instructions that are simultaneously in different pipeline stages, and take suitable counteractions accordingly. Typically, when an instruction is passing through the decode stage in the pipeline, the system checks if its input operands correspond to the same register which any other instruction already present in the pipeline is using as output. If this is true, there is a data dependency between the two instructions and some actions have to be taken. In this case, Register Forwarding must be activated: the input data for the second instruction, instead of coming from the register file, are directly taken from the stage where the first instruction produces them. In case instruction 1 is not yet in that stage, Pipeline Interlock is used. Pipeline Interlock implies that the pipeline is stalled until the first instruction reaches the stage in which the data is produced. At that moment, Register Forwarding is used to send the result of the first instruction to the stage where the second instruction needs its operands.

According to this working mechanism, there are different possible forwarding paths and their activation depends not only on the existence of some data dependency between two instructions but also on the stage in which the instructions produce/require the data.

The forwarding unit, implemented in the miniMIPS processor core occupies around 3.4% of the total number of gates of the processor, accounting for a total of 3,738 stuck-at faults.

a) ISA distillation

In order to target the forwarding unit, we decide to use most of the available instructions included in the miniMIPS ISA, since the module under consideration needs to exploit the data dependencies between program instructions. In this case, we do not rely to an atomic block specially devised for the considered unit.

The constraint library devised for the forwarding unit counts 53 different instructions, and it is described in an xml file containing 624 lines.

b) Concurrent test set generation

Once again, the evolutionary optimizer was set to create an initial population of 30 test programs containing on average 60 different instructions. The number of groups in the population is 20 and every group may include from 5 to 15 individuals.

The same *Perl* script used in the previous experience for the new genetic operator was also exploited in this experiment.

Figure 3 shows a part of the concurrent evolution of individuals and groups during the evolutionary run performed tackling the forwarding unit. The graph shows on the *X* axis the first 350 generations or steps, while the *Y* axis indicates the number of covered faults by both the best group (the highest line) and the best individual (the lowest line) at every generation.

Interestingly, the reader can notice that the group trend is always incremental, whereas it is different in the case of the individuals. In fact, it is possible to observe that around the 90^{th} generation, the best individual manages to cover 1,717 faults; however, it seems that this outstanding individual is not able to efficiently support a team, probably because its faults are already covered by less-performing individuals, and then some generations later it is discarded from the population. In any case, the coverage figures obtained by the best group outperforms along the evolution the best values reached by the individuals, showing that the cooperation scheme pursued by exploiting the Group Evolution is actually obtained at the end of the evolutionary process.

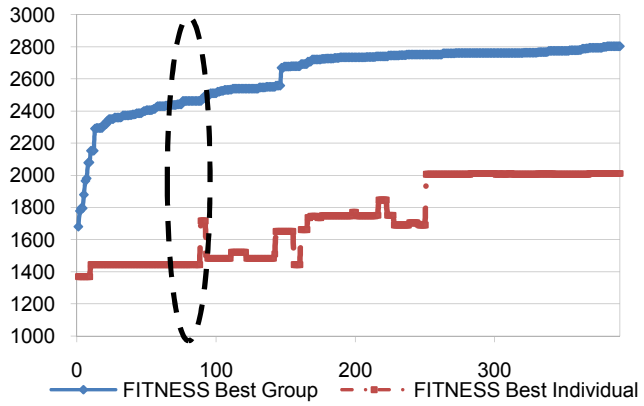

Figure 3. Evolutionary run for the forwarding unit

c) Final on-line setup

After 616 generations, taking about 94 hours of computational time in the same server described above, the evolutionary optimizer creates a group composed of 17 individuals able to cumulative reach 85.93% fault coverage on the miniMIPS forwarding unit. Summarizing, every test program belonging to the final best group contains about 59 instructions. Executing all programs in the best group takes 1,103 clock cycles, while each test program requires less than 100 clock cycles to be run.

```
…
ori $18, $14, 35284
sw $18, 1024($0)
sw $5, 1024($0)
lw $30, 0($0)
bne $7, $30, lK32K
beq $23, $29, LK325
…
```

Figure 4. Sample from one of the programs in the best group at the end of the evolutionary run for the forwarding unit

Figure 4 shows a small part from one of the programs composing the best group at the end of the evolution. Remarkably, it is possible to see different data dependencies between *ALU* instructions and *LD/SD* instructions (ori → sw), and also between *LD/SD* instructions and *BRN* instructions (lw → bne).

In order to experimentally validate our approach, we compared the obtained results in the forwarding unit with the ones obtained by a test set of programs tackling the whole miniMIPS processor core that achieves about 91% fault coverage against stuck-at faults. The test programs contained into the test set were developed following state of the art strategies such as [3]. However, the stuck-at Fault Coverage achieved on the considered forwarding unit reached only about 66%, thus proving that specific test programs are required for it.

5. Conclusions

The paper proposes an automatic strategy for the generation of functional test programs oriented to on-line testing of processor cores. The proposed strategy requires low human intervention, as well as little knowledge about the processor core or the modules under test.

In the paper, we tackled two common modules available in a pipelined processor core: the address calculation module and the forwarding unit. Both of the modules were approached using the proposed strategy, and the final coverage results experimentally corroborate the suitability of the proposed approach.

Preliminary results were obtained considering the stuck-at fault model, for an academic case study. They experimentally prove that the proposed technique is effective in achieving high fault coverage on the targeted modules, while maintaining essential characteristics for on-line testing.

References

[1] ISO/DIS26262 "Road vehicles – functional safety", 2009 (proposed std.)

[2] S. M. Thatte, J. A. Abraham, "Test Generation for Microprocessors", *Computers, IEEE Transactions on* , vol.C-29, no.6, pp.429-441, June 1980

[3] M. Psarakis, et al., "Microprocessor Software-Based Self-Testing", *IEEE Design & Test of Computers*, vol. 27, n. 3, pp.4 – 19, May-June 2010

[4] A. Paschalis and D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors", IEEE *Trans. Comput.-Aided Design Integr. Circuits Syst*., vol. 24, n. 1, Jan. 2005, pp. 88 – 99

[5] A. Merentitis, et al., "Directed Random SBST Generation for On-Line Testing of Pipelined Processors", *IEEE On-Line Testing Symposium*, 2008, pp. 273–279

[6] J. Shen, J. A. Abraham, "Synthesis of Native Mode Self-Test Programs", *Journal of Electronic Testing: Theory and Applications*, vol. 13, n. 2, October 1998, pp. 137-148

[7] H. Al-Asaad, et al., "Online BIST for embedded systems," *IEEE Design & Test of Computers*, vol. 15, issue 4, pp. 17–24, October 1988

[8] G. Squillero, "Artificial evolution in computer aided design: from the optimization of parameters to the creation of assembly programs", *Computing*, 2011, Volume 93, Numbers 2-4, Pages 103-120

[9] P. Bernardi, L. Ciganda, M. de Carvalho, M. Grosso, J. Lagos-Benites, E. Sanchez, M. Sonza Reorda,O. Ballan, "On-line software-based self-test of the Address Calculation Unit in RISC processors", *the 17th IEEE European Test Symposium (ETS)*, 2012, pp.1-6

[10] E. Sanchez, E. Schillaci, G. Squillero, *Evolutionary Optimization: the μGP toolkit*, Springer, 2011

[11] E. Sanchez, M. Sonza Reorda, and G. Squillero, "Test program generation from high-level microprocessor descriptions", in *System level Test and Validation of Hardware/Software Systems, ser. Springer Series in Advanced Microelectronics*, M. Sonza Reorda, Z. Peng, and M. Violante, Eds. Springer London, 2005,vol. 17, pp. 83–106

[12] E. Sanchez, G. Squillero, A. Tonda, "Group evolution: Emerging synergy through a coordinated effort", *Evolutionary Computation (CEC), 2011 IEEE Congress on* , pp.2662-2668

[13] A. Tonda, E. Lutton, G. Squillero, "Lamps: A Test Problem for Cooperative Coevolution", *Studies in Computational Intelligence*, 2012, Volume 387, Nature Inspired Cooperative Strategies for Optimization, pp. 101-120

[14] miniMIPS processor, available at http://opencores.org/project,minimips

[15] P. Bernardi, et al., "Fault grading of Software-Based Self-Test procedures for dependable automotive applications", *IEEE Design, Automation and Test in Europe Conference and Exhibition*, 2011