Chapter 9
Software-Based Self Testing of System Peripherals

Traditional test generation methodologies for peripheral cores are performed by a skilled test engineer, leading to long generation times. In this paper a test generation methodology based on an evolutionary tool which exploits high level metrics is presented. To strengthen the correlation between high-level coverage and the gate-level fault coverage, in the case of peripheral cores, the FSMs embedded in the system are identified and then dynamically extracted via simulation, while transition coverage is used as a measure of how much the system is exercised. The results obtained by the evolutionary tool outperform those obtained by a skilled engineer on the same benchmark. Preliminary results have been published in [127].

9.1 Introduction

A system-on-chip (SoC) can integrate into a single device one or more processor cores with standard peripheral memory and application-oriented logic modules. This high integration of many components leads to an increased complexity of the test process since it decreases the accessibility of each functional module into the chip. Thus, the ever increasing usage of such devices demands for cheap testing methodologies.

The Software-based Self-test (SBST), whereby a program is executed on the processor core to extract information about the functioning of the processor or other SoC modules and provide it to the external test equipment [87] meets this demands since: it allows cheap at-speed testing of the SoC; it is relatively fast and flexible; it has very limited, if any, requirements in terms of additional hardware for the test; it is applicable even when the structure of a core is not known, or can not be modified. Even though SBST is currently being increasingly employed, the real challenge of software-based testing techniques is to generate effective test programs.

Many SBST techniques have been developed for the test of microprocessor cores; traditional methodologies resort to functional approaches based on exciting specific functions and resources of the processor [151]. New techniques, instead, differ on

E. Sanchez et al.: Industrial Applications of Evolutionary Algorithms, ISRL 34, pp. 101–110.
springerlink.com

the basis of the kind of description they start from: in some cases only the information coming from the processor functional descriptions are required [41]; other simulation-based approaches require a pre-synthesis RT-level description [33] or the gate-level description [39].

Simulation-based strategies are heavily time consuming, thus, the use of RT-level descriptions to drive the generation of test sets is preferable to allow much faster evaluation. Relying on high-level models not only helps the user of the SoC to perform more simulations increasing the confidence in the generated tests, but is also of value to the manufacturer allowing early generation of a significant part of the final test set. Whereas the correlation between RT-level code coverage metrics (CCM) and gate-level fault coverage is not guaranteed in the general case, several RT-level based methodologies maximize the CCMs to obtain a good degree of confidence on the quality of the generated test set.

This paper describes the application of an evolutionary algorithm in test set generation process for different types of peripheral cores embedded in a SoC. Furthermore the generation process is fully automated and requires a very low human effort. The generation process is driven by the transition coverage on the peripheral's finite state machine (FSM) and by the RT-level Code Coverage Metrics (CCMs).

Exploiting the correlation between high-level and low-level metrics, during the generation process only logic simulation is performed allowing the reduction of the generation time. The results are finally validated running a gate-level fault simulation.

Results show that the combination of the FSM transition coverage and CCMs can effectively guide the test block generation and a high fault coverage can be achieved. Moreover, we show that the new approach makes the test generation process more robust, improving the relationship between high- and low-level metrics.

The rest of the chapter is organized as follows: section 2 recalls some background concepts in peripheral testing; section 3 outlines the methodology adopted for the generation of test sets and details the evolutionary tool. Section 4 introduces the experimental setup, describing the case study and presents the experimental results. Finally, section 5 draws some conclusions.

9.2 Peripheral Testing

9.2.1 Basics

A typical SoC is composed of a microprocessor core, some peripheral components, memory modules, and possibly customized cores. An external ATE is supposed to be available for test application: its purpose is to load a test program in the memory, start execution, and interact with the peripherals applying data to the input ports and collecting values from the outputs while the program is running.

To make effective use of the test setup both the test programs and the peripheral input/output data have to be specified; therefore, a complete set for testing peripheral cores is composed of some test blocks [17], defined as basic test units composed of two parts: a configuration and a functional part. The configuration part includes a program fragment that defines the configuration modes used by the peripheral, and the functional part contains one or more program fragments that exercise the peripheral functionalities as well as the data set or stimuli set provided/read by the ATE.

Researchers have long sought high-level methodologies to generate high quality test sets; this is possible only if a correlation between high-level metrics and gatelevel fault coverage exists. Differently from the general case, where the correlation is vague, in the case of peripheral cores this correlation actually exists. It is not complete but, as experimentally shown in [15], suitable for test set generation.

Therefore, an automatic methodology for the generation of test sets for peripheral cores that uses a high-level model of the peripheral in the generation phase is an interesting solution to overcome new testing issues on SoCs.

As mentioned in [17], traditional code coverage metrics suitable for guiding the development of the test sets for peripheral cores are: Statement coverage (SC), Branch coverage (BC), Condition coverage (CC), Expression coverage (EC), Toggle coverage (TC). Maximizing all the coverage metrics allows to better exercise the peripheral core. It is not possible to accept a single coverage metric as the most reliable and complete one [98]; thus different metrics must be exploited in order to guarantee better performance of the test sets [144].

9.2.2 Previous Works

An attempt to provide effective solutions for peripheral test set generation is presented in [17]; the process is performed by hand and mainly relies on the experience of a test engineer, who maximizes sequentially the various coverage metrics, generating one or more test blocks for every metric. This process is repeated until sufficiently high coverage values are obtained for all the chosen metrics. In [80] a pseudo-exhaustive approach to generate functional programs for peripheral testing was presented. The proposed method generates a functional program for each possible operation mode of the peripheral core in order to generate control sequences which would place the peripheral in all possible functional modes. The pseudo-exhaustive approach produces a large number of functional programs, since one has to be written for every operation mode.

In [6] the authors describe a generic and systematic flow of SBST application on two communication peripheral cores. The methodology achieves high fault coverage but needs a deep knowledge of the peripheral core leading to long test development time with a high human effort. In [15] the peripheral test set generation has been automated using an evolutionary algorithm, called μGP.

The test block generation was supported by the construction of couples of templates: one for program and the other for data generation. The evolutionary algorithm is used to optimize parameter values, leaving the structure of the test block fixed. The obtained results compare favorably with respect to the manually generated [17].

In [16] an improved version of the evolutionary algorithm has been described, able to optimize both the structure and the parameters. The same results as [15] are obtained with no need of the rigid templates used previously, reducing significantly the required generation time.

9.3 Proposed Approach

As stated above, traditional CCMs extracted at the RT-level do not, in general, show a tight correlation with gate-level fault coverage. Furthermore, the RT-level descriptions use, especially in the case of complex cores, many modules that interact among each other in order to perform the core functionalities. The traditional CCMs do not consider these interactions and only aim at maximizing the coverage metrics in each module. After the synthesis process, at the gate level, the distinction between modules of a core is less clear and therefore it is important to consider the interactions to enforce a correlation between high-level metrics and low level ones.

One way to model a system is to represent it with a FSM. Coverage of all the possible transitions in the machine ensures thoroughly exercising the system functions. Additionally, the use of FSM transition coverage has the additional advantage that it makes the interactions between functional modules in the peripheral explicit. Figure 9.3 sketches the proposed methodology.

The evolutionary approach generates test blocks starting from information about the peripheral core and the processor assembly syntax only. Every new test block generated is evaluated using a high-level simulator. The evaluation stage assigns a fitness to every individual. The procedure ends when a time limit is elapsed or when a steady state is detected, that is, a predefined number of test blocks are generated without any improvement of the coverage metrics. At the end of the evolutionary run a single test block is provided as output.

The sketched procedure is iteratively repeated to generate a complete test set. In the steps following the first one, the evaluation phase is modified in order to only take into account the additional coverage provided by the new test blocks. The rationale for this methodology is that in general it is not possible to completely solve the problem with one single test block. The end result of the process is a set of test blocks that cumulatively maximize the targeted coverage metrics.

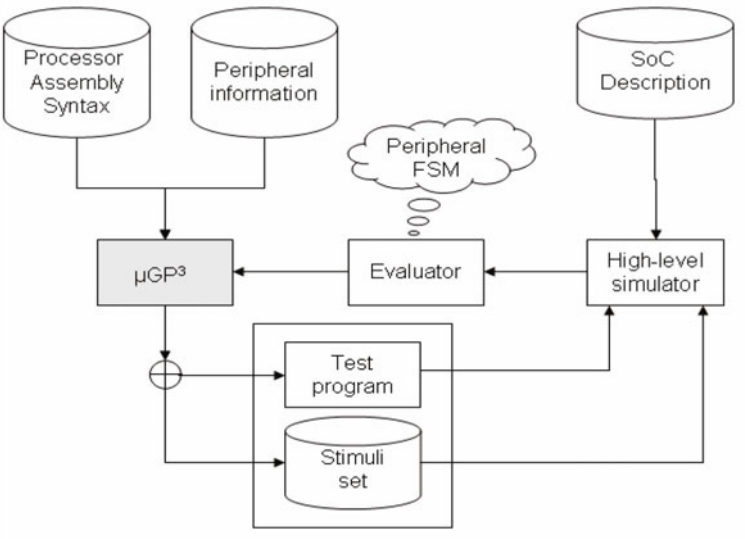

Fig. 9.1 Evolutionary generation loop.

9.3.1 Evolutionary Tool

For the automatic generation of the test blocks an evolutionary tool named ìGP3 [146] has been employed. μGP is a general-purpose approach to evolutionary computation, derived from a previous version specifically aimed at test program generation.

The tool is developed following the rules of software engineering and was implemented in C++. All input/output, except for the individuals to evaluate, is performed using XML with XSLT. The use of XML with XSLT for all input and output allows the use of standard tools, such as browsers, for inspection of the constraint library, the populations and the configuration options. The current version of the μGP comprises about 50,000 lines of C++ code, 113 classes, 149 header files and 170 C++ files.

Evolution Unit

μGP bases its evolutionary process on the concept of constrained tagged graph, that is a directed graph every element of which may own one or more tags, and that in addition has to respect a set of constraints. A tag is a name-value pair whose purpose is to convey additional information about the element to which it belongs, such as its name. Tags are used to add semantic information to graphs, augmenting the nodes with a number of parameters, and also to uniquely identify each element

during the evolution. The constraints may affect both the information contained in the graph elements and its structure. Graphs are initially generated in a random fashion; subsequently, they may be modified by genetic operators, such as the classical mutation and recombination, but also by different operators, as required by the specific application. The tool architecture has been specially thought for easy addition of new genetic operators as needed by the application. The activation probability and strength for every operator is an endogenous parameter.

The genotype of every individual is described by one or more constrained tagged graphs, each of which is composed by one or more sections. Sections allow to define a global structure for the individuals that closely follows the structure of any candidate solution for the problem.

Constraints

The purpose of the constraints is to limit the possible productions of the evolutionary tool, and also provide them with semantic value. The constraints are provided through a user-defined library that provides the genotype-phenotype mapping for the generated individuals, describes their possible structure and to define which values the existing parameters (if any) can take.

Constraint definition is left to the user to increase the generality of the tool. The constraints are divided in sections, every section of the constraints matching a corresponding section in the individuals. Every section may also be composed of subsections and, finally, the subsections are composed of macros.

Constraint definition is flexible enough to allow the definition of complex entities, such as the test blocks described above, as individuals. Different sections in the constraints, and correspondingly in the individual, can map to different entities. In this specific case the constraints define three sections: a program configuration part, a program execution part and a data part or stimuli set. The first two are composed of assembly code, the third is written as part of a VHDL testbench. Though syntactically different, the three parts are interdependent in order to obtain good solutions. Fitness. Individual fitnesses are computed by means of an external evaluator: this may be any program able to provide the evolutionary core with proper feedback.

The fitness of an individual is represented by a sequence of floating point numbers optionally followed by a comment string. This is currently used in a prioritized fashion: one fitness A is considered greater than another fitness B if the n-th component of A is greater than the n-th component of B and all previous components (if any) are equal; if all components are equal then the two fitnesses are considered equal.

Evolutionary Scheme

The evolutionary tool is currently configured to cultivate all individuals in a single panmictic population, although it can be configured to use an island model. The

population is ordered by fitness. Choice of the individuals for reproduction is performed by means of a tournament selection; the tournament size τ is also endogenous. The population size ì is set at the beginning of a run, and the tool employs a variation on the plus $(\mu + \lambda)$ strategy: a configurable number λ of genetic operators are applied on the population. Since different operators may produce different number of offspring the number of individuals added to the population is variable. All new unique individuals are then evaluated, and the population resulting from the union of old and new individuals is sorted by decreasing fitness. Finally, only the first μ individuals are kept.

To promote diversity, the individuals genetically equal to already existing ones, called clones, may have their fitness scaled by a fixed value in the range [0.0,1.0]. The possible termination conditions for the evolutionary run are: a target fitness value is achieved by the best individual; no fitness increase is registered for a predefined number of generations; a maximum number of generations is reached.

At the end of every generation the internal state of the algorithm is saved in a XML file for subsequent analysis and for providing a minimal tolerance to system crashes.

9.3.2 Evaluator

The proposed approach is based on modeling the entire system as a FSM which is dynamically constructed during the test generation process. Thus, differently from other approaches, the FSM extraction is fully automated, and requires minimum human effort: the approach only requires the designer to identify the state registers in the RT-level code; every global state in the peripheral represents a possible configuration of values of all the state registers. Thus, whenever a state register in any module changes its value, also the global state of the peripheral is affected. Given the dynamic nature of the FSM construction, it is not possible to assume known the maximum number of reachable states, not to mention the possible transitions. For this reason it is impossible to determine the transition coverage with respect to the entire FSM.

As experimentally demonstrated [98], maximizing more than one metric usually leads to better quality tests. Thereby, the simulation-based method proposed here exploits the FSM transition coverage, that enforce a maximum interaction between peripheral modules, and all the available CCMs to thoroughly exercise the peripheral functionalities.

The implemented evaluator collects the output of the simulation and dynamically explores the FSM; it assesses the quality of the test block considering the transition coverage on the FSM and the CCMs.

The fitness fed back to the evolutionary tool is composed of many parts: the FSM transition coverage followed by all the others CCMs (SC, BC, CC, EC, TC). As we mentioned before the metrics are considered in order of importance. In this way it is

possible, during the generation process, to select more thoroughly those test blocks that are able to better excite the peripheral.

9.4 Experimental Analysis

9.4.1 Test Case

The benchmark is a purposely designed SoC which includes a Motorola 6809 microprocessor, a Universal Asynchronous Receive and Transmit (UART), a Peripheral Interface Adapter (PIA), a Video display unit (VDU) and a RAM memory core. The system derives from one available on an open source site [112]. The methodology is used to test the UART, the PIA and the VDU in the targeted SoC.

The peripherals are described at RT-level in VHDL code and are composed of different modules. The SoC was synthesized using a generic home-developed library.

Table 9.1 Implementation characteristics

Description	Measure	PIA	VDU	UART
RT-level	statements	149	153	383
	branches	134	66	182
	condition	75	24	73
	expression	0	9	54
	toggle	77	199	203
Gate level	Gates	1,016	1,321	2,247
	Faults	1,938	2334	4,054

Table 9.1 shows details of the targeted peripherals, including information at high and low level. Rows labeled with RT-level present CCM information while the remaining rows illustrate the number of gates counted on the synthesized devices and the number of collapsed faults for the stuck-at model, respectively.

At the end of the generation process, some gate-level fault simulation were performed only to validate the proposed methodology; the gate-level fault coverage figures reported in the following sections target the single stuck-at fault model.

9.4.2 Experimental Results

All the reported experiments have been performed on a PC with an Athlon XP3000 processor, 1GB of RAM, running Linux.

The algorithm parameters for the evolutionary experiments are the same both when targeting only the CCMs, and when the number of transitions in the FSM is also taken into account: for the PIA and the VDU experiments, $\mu = 50$ and $\lambda = 70$; and as the UART is more complex than the PIA the evolutionary parameters were set to perform a lower number of simulations: μ was set to 30 and λ to 40.

In order to provide the reader with a reference value, we recall that the fault coverage obtained by the manual approach presented in [17] is 80.96% for the UART and 89.78% for the PIA.

Table 9.2 summarizes the results obtained for the targeted peripherals, reporting the number of FSM transitions covered, the high-level CCMs and the stuck-at fault coverage (FC) in percentage. The reader should note that the value of traditional CCMs are expressed as absolute values (instead of percentages).

Table 9.2 Results for considered peripherals

	PIA	**VDU**	**UART**
FSM Transition	115	191,022	142
Statement	149	153	383
Branch	129	66	180
Condition	68	23	72
Expression	0	9	51
Toggle	77	191	203
FC(%)	91.4	90.8	91.28

For every peripheral considered the methodology is able to reach a good value of gate-level fault coverage. In the case of the VDU the number of transition is very high; this is due to the state registers that hold the current position on the screen.

To experimentally demonstrate that the use of the FSM transition coverage is essential to strengthen the correlation between high an low level metrics 100 experiments on the UART are performed, using both the evolutionary approach presented in [16] and the generation process detailed above.

Table 9.3 Comparison between the two methodologies

		FSM	**SC**	**BC**	**CC**	**EC**	**TC**	**FC**
[16]	Average	NA	381.8	178.7	70.7	50.7	201.3	84.8
	std.dev.	NA	0.36	0.39	0.30	0.32	0.40	6.37
New methodology	Average	141.0	382.2	179.3	71.8	50.8	202.2	90.9
	std.dev.	1.49	0.28	0.33	0.22	0.24	0.36	1.10

Table 9.3 reports a comparison between the results of the experiments performed following the methodology presented in [16] and the current one; the table illustrates

the average and standard deviation of the different CCMs and of the stuck-at fault coverage (FC). In all cases the CCMs are very near to the absolute maximum, and both methodologies lead to small standard deviations on the considered metrics. In the first case, however, the standard deviation in the fault coverage of each test set is relatively high. Although the methodology obtains good results, it is not as robust as desirable, and the obtained solution may not exhibit the expected quality.

Using the new methodology the average fault coverage is increased by more than 6% and, more importantly, the standard deviation of the fault coverage is dramatically reduced. This clearly shows that the robustness of the methodology is increased, and solutions of consistent quality can be obtained.

Table 9.4 Comparison between the two methodologies

	FC	TGEN	TAPP	Size
[16]	90.7	5.1	28,842	1,953/72
New Methodology	91.3	2.2	32,762	2,345/87

Table 9.4 synthetically reports a comparison between the two methodologies in the case of the UART, highlighting the obtained fault coverage (FC) in percentage, the average generation time (TGEN) expressed in hours, the average application time (TAPP) in clock cycles, and the average size of the test sets, reported as program bytes and data bytes. The results clearly show that the new methodology outperforms the previous one in terms of fault coverage and generation time. The latter, in particular, is less than a half with respect to the previous methodology, highlighting the efficiency of the new approach.

Other approaches [80][6] to peripheral test are not directly comparable with our methodology since they are referred to different devices, although their complexity and the results are similar to the devices analyzed here. Furthermore, our methodology only needs RT-level simulation and does not need the time-expensive fault-simulations.

9.5 Conclusions

In this chapter, a successful application of the evolutionary tool for the generation of sets of test blocks for different types of peripheral modules in SoCs driven by the FSM transition coverage and the high-level CCM has been described.

The evolutionary tool is able to generate test blocks where the relation between high-level coverage metrics and low level one is much stronger; this better relation has been experimentally demonstrated with a experimental analysis where many test blocks are generated and evaluated.

The experimental results on different type of peripheral cores, communication peripherals and VDU controller, show the effectiveness of the proposed methodology.