Coupling EA and High-level Metrics for the Automatic Generation of Test Blocks for Peripheral Cores

L. Bolzani	E. Sanchez	M. Schillaci	G. Squillero
Politecnico di Torino	Politecnico di Torino	Politecnico di Torino	Politecnico di Torino
Corso Duca degli Abruzzi, 24	Corso Duca degli Abruzzi, 24	Corso Duca degli Abruzzi, 24	Corso Duca degli Abruzzi, 24
Torino, Italy	Torino, Italy	Torino, Italy	Torino, Italy
+39 0115647091	+39 0115647091	+39 0115647091	+39 0115647091
leticia.veirasbolzani@polito.it	edgar.sanchez@polito.it	massimiliano.schillaci@polito.it	giovanni.squillero@polito.it

ABSTRACT
Test of peripheral modules has not been deeply investigated by the research community. When embedded in a system on chip, however, peripherals pose accessibility problems that may make traditional test approaches ineffective. In this paper an evolutionary methodology, based upon coverage metrics at high-level, is described to automatically generate test sets for peripheral modules in a SoC. A general-purpose evolutionary tool, able to cultivate composite individuals, has been developed and is used for the test set generation. This tool is described and its basic concepts explained. The method compares favorably with results obtained by hand.

Categories and Subject Descriptors
B.8.1 [**Performance And Reliability**]: Reliability, Testing, and Fault-Tolerance

General Terms
Algorithms, Design, Experimentation.

Keywords
Peripheral testing, test programs, test blocks, SBST.

1. INTRODUCTION
Modern methodologies in industrial production of electronic circuits attempt to reuse and integrate as much as possible mature devices. The goal is that of decreasing both production costs and time-to-market. Since the mid '90s, the *System-on-Chip* (SoC) paradigm is successfully being employed for integration and reuse of electronic devices. Indeed, SoC based applications are now used in practically every consumer electronics device around the world.

Generally, a SoC integrates at least one processor core, some peripherals devices, and a few memory cores into a single chip. While simplifying the design and verification phases, the SoC architecture increases the complexity of the test process because it decreases the accessibility of each functional module into the chip. Thus, the increasing use of SoCs is leading to new issues on production testing methodologies.

Different software-based and hardware-based techniques have been proposed to test SoCs and the cores they embed. Hardware-based techniques are commonly based on scan chains or built-in self-test (BIST) insertion [13]. These methods are easy to implement and scale well with increasing complexity; however, they may generate high area overhead, impair performance, and require unacceptably long application times. In addition, at-speed testing of the SoC is not always economically feasible using scan chains, but test application with a reduced clock speed considerably decreases the efficiency of test vectors [8].

One of the less intrusive and more promising techniques is based on exploiting the processor core embedded into the SoC to execute a special program, which is able to test the processor, and other cores accessible by it [1]. This technique, called *Software-based Self-Test*, or SBST, has several advantages with respect to traditional hardware-based ones: 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. As stated above, SBST is based on the execution of a test program previously loaded in an internal memory. This test program is not intended to perform any functionality but only to test a specific module or core internal to the SoC. The result produced by the system is checked by monitoring what is produced on specified output ports or memory variables. On the downside, SBST raises the issue of how to generate effective test programs.

Appropriate test programs have been successfully exploited not only for manufacturing tests, but also for design validation or verification, incoming inspection, and on-line testing [2].

SBST techniques have been largely exploited on testing microprocessor cores; traditional methodologies resort to functional approaches based on exciting specific functions and resources of the processor [3]. New techniques, instead, differ on the basis of the kind of description they start from: in some case only the information coming from the processor functional description are required [1]; in other cases, a pre-synthesis RT-level description is required [6]; most often, the gate-level description is exploited to generate an efficient set of test programs [5], especially if aimed at manufacturing test.

Peripheral testing had not received the same attention that processor cores testing did. In fact, testing peripherals embedded in a SoC is an overlooked problem. However, the reduced accessibility of peripheral modules inside a chip may make the use of traditional testing methodologies, aimed at stand-alone peripheral chips, ineffective.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GECCO'07, July 7–11, 2007, London, England, United Kingdom.

Copyright 2007 ACM 978-1-59593-697-4/07/0007...$5.00.

In this paper, we face the problem of testing peripheral components in a SoC resorting to a software-based approach. The presented methodology exploits RT-level Code Coverage Metrics (CCMs) on the generation of the test sets.

The generation problem is approached resorting to the new implementation of an evolutionary algorithm (EA). The described methodology is applied to automatically generate *test blocks* for two peripheral cores embedded in a SoC that includes a Motorola 6809 processor core. For the sake of completeness, this paper compares the automatic approach with the manual one described in [12].

The rest of the paper is organized as follows: section 2 recalls some background concepts, describes the RT-level coverage metrics suitable for test generation, and introduce some considerations regarding the generation process; section 3 outlines the methodology adopted for the generation of test set for peripheral testing. Section 4 introduces the experimental setup, describing the case study and detailing the workflow. Additionally, this section brings implementation details on the test generation process. Finally, section 5 draws some conclusions.

2. PERIPHERAL TESTING

Generally speaking, a basic SoC is composed of a microprocessor core, some peripheral components, a few memory modules, and possibly customized cores. Figure 1 shows the basic block diagram of a typical SoC. An external Automatic Test Equipment (ATE) is supposed to be available for test application; it may interact with the SoC through the interface modules. Additional ports may be available to the ATE to access the SoC (e.g., an IEEE 1149.1 TAP port), but methodologies that use them are out of the scope of this paper.

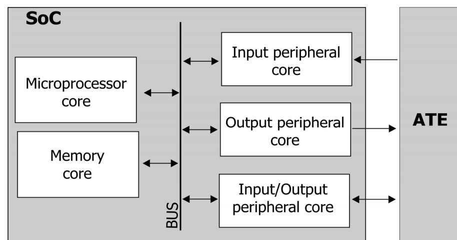

Figure 1. Block diagram of a SoC

Test program generation for SBST may benefit from cross-fertilization with the area of design validation. In fact, design validation during the pre-silicon phase is also performed resorting to suitable test programs. In [6] test programs are composed of dynamic sequences of code segments and are generated for simulation at the RT-level to verify application behaviors and detect hardware design bugs; once generated, these programs can be fruitfully exploited for later manufacturing test. However, the authors do not state a relationship between the test programs generated at high level and low-level fault coverage.

In [7] the authors define a fault model at high level strongly correlated with the traditional low-level stuck-at fault. The presented methodology exploits the so called *observability-enhanced statement coverage metric* and models single stuck-at bit faults on all assignment targets of the executed statements that respect a defined set of rules. These rules are defined to identify redundant faults in the fault list in order to increase the correlation between RT-level (high level) and gate-level fault coverage (low level).

Despite the efforts, the correlation between high-level and low-level metrics still remains vague in the general case, especially when large combinational blocks exist in the design, whose testability can barely be forecast when resorting to high-level metrics, only.

Regarding test application, [4] and [5] describe suitable architectures able to support SBST. The approaches require an easily accessible RAM memory of sufficient size available into the SoC. An ATE is in charge of loading the test program into the memory when required, and the processor core is forced to execute it. Some mechanism must also be available to retrieve and check the results. Test execution is always performed at-speed, independently on the speed of the mechanisms used for loading the RAM and checking results: low-cost ATEs can thus be exploited, greatly reducing the cost for test.

Peripheral testing problem has not been as deeply investigated as microprocessor testing. Some important reasons could motivate this fact: peripheral cores are usually smaller than processors by at least one factor of magnitude; stand-alone peripherals are reasonably easy to test; when the testing methodology is based on hardware insertion, the area overhead required to test peripherals is negligible. However, testing of peripheral cores embedded in a SoC leads to new issues even for hardware-based approaches: first of all, peripherals become less accessible due to the SoC structure; if the SoC design uses multiple clocks, scan insertion is complicated and may not obtain the expected coverage; finally, if SoC architecture exploits internal tristate buses to share resources among different modules, this requires the definition of quite complex constraints to properly generate scan vectors.

Therefore, a software-based strategy for peripheral testing that uses a high-level model of the peripheral in the generation phase could be a suitable solution to overcome new testing issues on SoCs. Remarkably, there is special value in devising techniques starting from high-level descriptions, since they can be exploited even by soft-cores users (and not only by core developers) and make test reuse easier. On the other side, this is only possible if a well-known relationship exists between high-level and low-level metrics.

In order to support the generation process of the test sets for peripheral cores, let us assume that a complete test set for peripheral cores is composed of some *test blocks* 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 provided/read by the external test machine. Figure 2 shows the conceptual scheme of a test block.

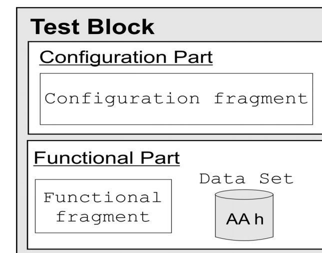

Figure 2. Conceptual scheme of a test block

Test set generation process exploits both the CCMs and the basic functionalities of each peripheral core to drive the generation process.

The code coverage metrics suitable for guiding the development of the test sets for peripheral cores embedded in a SoC are listed in the following (and their definition briefly summarized for sake of completeness).

- *Statement coverage (SC)* is the most basic form of code coverage: statement coverage is a measure of the number of executable statements within the model that have been exercised during the simulation run. Executable statements are those that have a definite action during runtime and do not include comments, compile directives or declarations. Statement coverage counts the execution of each statement on a line individually, even if there are multiple statements on that line.
- *Branch coverage (BC)* reports whether Boolean expressions tested in control structures (such as the if- and while-statement) are evaluated to both true and false. The entire Boolean expression is considered one true-or-false predicate regardless of whether it contains logical-and or logical-or operators. Branch coverage is sometimes called decision coverage.
- *Condition coverage(CC)* reports the true or false outcome of each Boolean sub-expression, separated by logical-and and logical-or if they occur. It is an extension of branch coverage. Condition coverage measures the sub-expressions independently of each other.
- *Expression coverage (EC)* is the same as condition coverage, but instead of covering branch decisions it covers concurrent signal assignments. It builds a focused truth table based on the inputs to a signal assignment using the same technique as condition coverage.
- *Toggle coverage (TC)* reports the number of bits that toggle at least once from 0 to 1 and at least once from 1 to 0 during the logical simulation of the device. At the RT-level registers are targeted and, since RT-level registers correspond to memory elements with an acceptable degree of approximation, the toggle coverage is an objective measure of the activity of the design.

According to the above definitions, maximizing all coverage metrics helps to better exercise the peripheral cores. However, it is essential to carefully choose the set of metrics to be maximized to reduce redundant efforts on the test set generation.

Many authors hold that it is not possible to accept a single coverage metric as the most reliable and complete one [9]; thus, a coverage of 100% on any particular metric can hardly guarantee a 100% fault coverage. Nowadays, thanks to modern logic simulators features, different metrics can be exploited to guarantee better performance of the test sets. Therefore, the test set generation trend is to combine multiple coverage metrics together to obtain better results. Consequently, it is extremely useful to complete the test sets by reaching complete coverage on different metrics [10].

Among the listed metrics, statement coverage deserves special attention because this one has been considered until now the most popular coverage metric to evaluate the effectiveness of test sets. In fact, statement coverage could be considered as the first metric to be maximized since this is the single most effective measure of design utilization during performance tuning. It is, however, not reliable for test coverage purposes.

The selection of the sequence of other coverage metrics must be based on an analysis of the RT-level description of the targeted peripheral. As mentioned before, the first metric to be considered is normally the statement coverage; the successive metrics are strongly related to the description style used to write the peripheral model at high-level. For example, if the high-level description is based on a *case* or *if-then* structure the second coverage metric to be targeted is the branch metric. On the other side, if the description uses a set of variables computed on logical operations and Boolean expressions, we should consider the expression coverage as the most suitable metric.

Let us consider the following VHDL code fragment to clarify the results that can be obtained using different metrics:

```
CONTROL: process (INPUT)
begin
        case INPUT is
                when (3 | 5) =>
                        Z <= A and B;
                when 1 =>
                        Z <= A or B;
                when others =>
                        null;
        end case;
end process CONTROL;
```

Let assume that there are two available sets of patterns for the control variable INPUT ranging from 0 to 7:

```
set1 = {(0), (1), (3)}
set2 = {(0), (1), (3), (5)};
```

For both input sets, the statement coverage and the branch coverage equal 100%; however, `set2` reaches a higher value in the condition coverage, since this set of stimuli thoroughly exercises the statement "`when (3 | 5) =>`".

2.1 Traditional approaches

The manual method [12] is based on the experience of a test engineer and resorts to a preliminary evaluation of the peripheral high-level description. The test engineers must also select the order in which the CCMs will be maximized. The order followed in the description of the CCMs is acceptable in most situations.

The general scheme for manual test-set generation is shown in figure 3. In the first step, an initial test set is generated targeting to maximize the statement coverage; the first test block is only based on functional information about the targeted peripheral. Then, the generated test block is simulated using a RT-level description, gathering the first code coverage metrics figures.

Following the selection of the most suitable metric (according to the considerations of the previous sub-section) a new set of test blocks looking for the saturation of the considered metric is devised. Once the first coverage metric is saturated, another one is

tackled in order to increase the testing capabilities of the generated test set [10].

This process is repeated until sufficiently high coverage values are obtained for all the chosen metrics. Remarkably, it must be taken into consideration that in some cases the metrics chosen as critical may also differ, as well as the number of considered CCMs for sufficient code coverage.

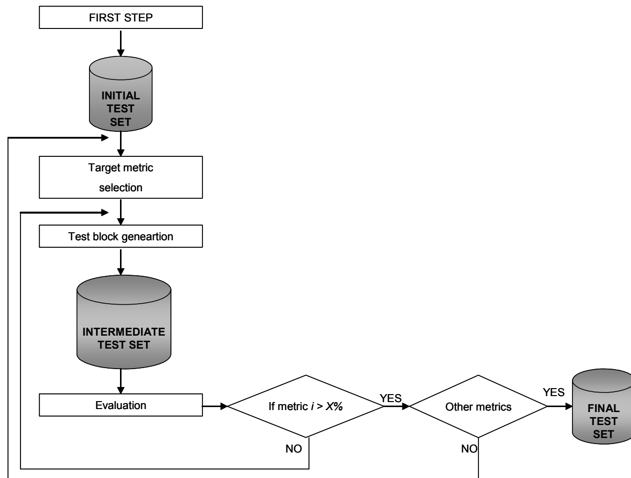

Figure 3. Test set generation scheme

The target value for each code metric is denoted by *X%* in figure 3. It should be carefully identified based on the description style: 100% code coverage is not always reachable, as explained in [9].

Differently from the manual method, in [11] 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 method exploits the capacity of the embedded microprocessor to test peripheral cores at-speed. The pseudo-exhaustive approach was employed to test an *Intel 8251 Universal Synchronous Asynchronous Receiver Transmitter* peripheral core and about 68% of fault coverage was obtained.

The pseudo-exhaustive approach [11] gives rise to a large number of functional programs, since one has to be written for every operation mode; this implies a large application time and a considerable memory occupation.

3. PROPOSED APPROACH

Fault coverage is a distinctly nonlinear function of the input patterns for most digital circuits, and peripheral modules are no exception. Moreover, the cases to cover for effective testing may be too numerous for a test engineer to handle without support. Although humans may cleverly identify critical corner cases for the operation of a circuit, in fact, they are not generally renowned for reliably considering large numbers of alternatives and selecting suitable subsets from those. These considerations make the test pattern generation amenable to the use of evolutionary algorithms.

For every peripheral core into the SoC a test set is devised by using the high-level description of the component. It is worth noting that in the generation process only the CCMs have been used as feedback information. A fault simulation is subsequently performed to validate the generated test sets.

Since the testing methodology presented here is based on a software-based approach, it is necessary to generate suitable test blocks containing program fragments able to not only configure, but also to exercise the peripheral core while working in normal mode operation. Additionally, it must be clear that the chosen data to be received/transmitted by the peripheral may also impact the final testing results. These data could be included into the test program or provided by the external test machine. Therefore, both test programs and data must be carefully combined in order to obtain high coverage figures.

3.1 Evolutionary tool

For the automatic generation of the test blocks an evolutionary tool named μGP^3 has been employed. μGP^3 is a general-purpose approach to evolutionary computation, derived from a previous version specifically aimed at test program generation. The problem of test program generation had already been tackled in the past by some of the authors [14] [15]. The efforts led to the implementation of a versatile evolutionary tool with a strong emphasis on assembly program generation, named μGP. This tool was successfully employed for test program generation, and proved versatile enough for such diverse activities as *corewar* program evolution, antenna array optimization, expression evolution.

During several years the successive versions of μGP have been improved with numerous features to improve its performance. These include aging and eventual death of the individuals, a configurable size for the elite (individuals that never age), self-adaptation of many evolutionary parameters such as operator activation probability and strength and of tournament size; an entropy fitness hole has also been added, as has clone detection and optional scaling/extermination.

However, it reached a development point in which it was difficult to extend it and to work collaboratively on its enhancement; moreover, there was the need for a more comprehensive tool that allowed tackling different problems and experimenting evolutionary techniques untried before, such as support for multiple populations, possibly with different topologies, and for true multi-objective optimization.

These considerations led to the decision to completely redesign the tool and implement it from scratch, basing its development on sound software engineering techniques and taking advantage of existing software and established standards. The tool was thus implemented in C++, and all input/output, except for the phenotype of the individuals to evaluate, is performed using XML with XSLT.

The current version of the μGP^3 comprises about 40,000 lines of C++ code, 85 classes, 108 header files and 103 C++ files[1].

[1] μGP^3 is available under the terms of GNU public license from SourceForge (http://ugp3.sourceforge.net/)

3.2 Evolutionary concepts

3.2.1 Evolution unit

μGP[3] 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. 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.

3.2.2 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. The constraints may specify every section as compulsory, meaning that the sections have to exist in every individual, or optional. Every section may also be composed of *subsections*: for each a minimum and a maximum number may be specified. Finally, the subsections are composed of *macros*, of which a minimum and maximum number can also be set.

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 such as the configuration part of the program, the functional part, and the external data. No special tweaking of the tool is needed to handle these different concepts.

3.2.3 Fitness

Individual fitnesses are computed by means of an external evaluator: this is usually a script that runs a simulation using the individual as input and collects the results, but may be any program able to provide the evolutionary core with proper feedback. This complete decoupling between the evolutionary engine and the fitness evaluator makes it possible to use μGP[3] with most existing simulation tools.

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. The two value sequences may then be considered symbols in a string, and the fitnesses may be thought of as such strings, which are then compared lexicographically. For uniform comparison all those strings have to be equal length, meaning that the number of values in the individual fitness has to be specified before every run. It may, however, vary between one run and the next.

3.2.4 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 μ 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 ensure that only different individuals are evaluated a hash is computed for each individual, only comparing individuals for equality if their hashes are the same. To promote diversity, individuals that are 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; the evolution process is executed for the maximum number of generations.

At the end of every generation the internal state of the algorithm is saved in XML format for subsequent analysis. This also provides a minimal measure of tolerance to system crashes.

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 [16].

4. CASE STUDY

In order to experimentally demonstrate the suitability of the presented methodology, a SoC was implemented including a Motorola 6809 microprocessor, a serial communication peripheral (*Universal Asynchronous Receive and Transmit*, UART), a parallel communication peripheral (*Peripheral Interface Adapter*, PIA) and a RAM memory core. The system used derives from one available on an open source site [17]. Figure 4 shows the schematic view of the considered SoC.

The high-level description of every component was written in VHDL at RT-level; the whole SoC is described in about 12,000 lines of code, and the synthesized circuit contains approximately 20,000 equivalent gates. The following table summarizes implementation characteristics of the peripheral cores presented in the SoC, and it includes some facts at high and low levels of description.

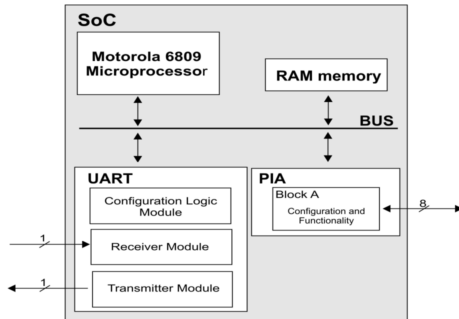

Figure 4. Case study architecture

Table 1 shows details of both peripherals the PIA and the UART, including information at high and low-level. Rows 2 to 6 present CCMs information, such as the number of statement of the RT-level descriptions of the peripherals. The reader should be aware that in the case of the PIA there are no available expressions to be considered; this is due to the specific design style used to describe the peripheral at RT-level. At low level (i.e., at gate level), rows 7 and 8 illustrate the number of gates counted on the synthesized devices and the number of collapsed faults for the stuck-at model, respectively.

Table 1. Implementation characteristics

Description	measure	PIA	UART
RT-level	statements	149	383
	branches	134	182
	condition	75	73
	expression	N/A	54
	toggle bits	77	203
Gate level	gates	1,016	2,247
	faults	1,938	4,054

Both the PIA and the UART peripherals can be configured to work in two functional modes with different communication schemes.

The PIA can be used with the following characteristics:

- polling or interrupt mode
- parallel data communication (transmit and receive) with different control schemes.

In the case of the UART, the following configuration modes can be used:

- polling or interrupt mode
- serial data communication (transmit and receive) with different data bit numbers, with or without parity, and with 1 or 2 stop bits
- serial transmit and receive using different communication rate ratios.

To more thoroughly assess the proposed methodology, a test set for each peripheral core was generated resorting to both the manual and EA-based methodology. In both cases, the generation process uses only, as feedback or fitness values, information extracted from the RT-level simulation of the SoC (i.e., the high level metrics described in section 2.2). The manual method has been completely performed by an expert test engineer.

The fault coverage figures reported in the following sections target the stuck-at fault model, and the gate-level fault simulations required were performed at the end of the generation processes in order to bring the reader the possibility to clearly assess the proposed methodology.

4.1 Manual methodology setup

The test set for each peripheral core is composed of many test blocks, each including two elements: the first one (configuration part) defines the operation mode to be adopted by the peripheral; the second element (functional part) exploits the peripheral functionalities (i.e., transmitting/receiving data).

Based only on the functional analysis of the targeted communication peripheral, a first test block was developed. A RT-level simulation was then performed to collect the initial code coverage values.

The development of each successive test block is based on analyzing the VHDL code style to identify the most suitable metric, and on its optimization.

The analysis of the PIA's VHDL description resulted in choosing the statement coverage metric as the most suitable to be targeted. This choice is justified by the fact that in the considered code the dominant structures can be categorized as statements. This means that most of the statements are not associated with any structure of the branch or condition types. When we had achieved satisfying statement coverage we could also observe acceptable percentages of coverage in the other metrics.

Differing from the PIA, the UART required explicitly targeting several metrics to achieve useful results. Since the considered VHDL description models use several *case* instructions, the most suitable metric to be targeted after the statement is the branch coverage. As stated in [10], maximizing the branch coverage leads to almost fully maximized statement coverage, also.

After the maximization of the branch metric, we analyzed the code coverage values of the other metrics and we observed that the value of the condition coverage was also sufficiently high; on the other side, the values obtained on the expression coverage were still relatively low.

We found that the reason for this was mainly due to the fact that the transmission control is performed by reading the status register. The code lines performing this control can be classified as expressions. For this reason, the third metric we considered was the expression coverage. New test blocks were thus devised and iteratively improved until the expression coverage reached a reasonably high value.

4.2 EA based methodology setup

To adequately handle the requirements on peripheral testing, two constraints libraries were devised, one for each peripheral core. To quickly generate satisfactory test blocks it has been decided to

reduce as much as possible the size of the search space for the evolutionary tool. Basically, the constraints libraries were implemented using two sections: the first one concerning program generation, and the second one data generation.

Program sections include two subsections, one for each operation mode: polling and interrupt. In both cases, the generated macros resemble the structure of a hand-written program: the initial section configures the peripheral core via a small number of fixed initialization sequences, whereas the second one applies or asks some data. It is clear that in the case of interrupts, the second part of the program is adequately placed in memory in order to correctly resolve an interrupt request. In the end a fixed observation sequence is provided.

The configuration sections are multiple because the final goal is to generate a set of test blocks that collectively test the peripheral, not just one that does everything. Test blocks are sequentially generated until their cumulative coverage for all the listed CCMs is satisfactory.

Data sections, on the other hand, contain some macros able to bring the peripheral core with input data, as well as with appropriate control signals in order to exercise the peripheral inputs. External data does not undergo any arbitrary restriction. Additionally, a waiting macro was included in order to better match timing constraints.

As mentioned before, the evolutionary tool used as fitness values a sequence of floating point values representing the CCMs figures obtained by RT-level simulation. All the available coverage values were fed back to the evolutionary tool in the order specified above, in order to simultaneously optimize all of them. The experiments were launched using a multi-run scheme,. Thus, once the steady state was reached a new run is launched excluding the elements covered during the previous experiments. This scheme is followed until no further progress is possible.

4.3 Comparison of the results

The following tables compare the results obtained for each peripheral core following the manual and EA-based methods. Tables illustrate the percentage value of the different coverage metrics obtained at each step of the elaboration as well as the attained fault coverage (FC). For clarity, the intermediate results of the EA-based experiments are reported. Since they are only used to validate the evolutionary methodology, in the case of the manual method, only the final results are reported.

Table 2. PIA – manual method

Step	SC	BC	CC	EC	TC	FC
8	**100.0**	**96.9**	**89.3**	**N/A**	**100.0**	**89.78**

Applying the manual methodology to generate test blocks for the PIA, eight different steps were required to obtain a satisfactory coverage on the CCMs. The 8 resulting test blocks are composed of 8 programs sizing about 200 bytes. Additionally, the final data set contains 19 input data and the duration of the complete set of test blocks requires 800 clock cycles.

For the experiment a constraints library was written with about 200 lines of XML. The evolutionary parameters were μ=50 and λ=70, whereas the steady state parameter was set to 10 generations for each run.

Table 3. PIA – EA based method

Step	SC	BC	CC	EC	TC	FC
1	81.1	73.4	73.3	N/A	94.8	80.50
2	93.0	86.7	88.0	N/A	100.0	88.67
3	97.2	92.2	89.3	N/A	100.0	89.45
4	99.3	96.1	89.3	N/A	100.0	89.73
5	**100.0**	**96.9**	**90.7**	**N/A**	**100.0**	**90.20**

The automatic method left us with 5 test blocks to test the PIA. The size of the generated programs is about 480 bytes, and the data set contains 80 input data. To apply the complete sequence of test blocks 5,800 clock cycles are required. It is interesting to see that the final values reported by both tables are almost identical; however, in the case of the EA-based method the FC is higher and the CC metric has been saturated.

Table 4. UART – manual method

Step	SC	BC	CC	EC	TC	FC
7	**95.5**	**92.9**	**97.3**	**72.2**	**89.2**	**80.96**

Manually written programs amount to about 250 bytes. The final data set contains 10 input data and the duration of the complete set of test blocks involves about 5,000 clock cycles. The manual methodology produced 7 test blocks. The manual method was not able to reach higher coverage figures because of time constraints.

Table 5. UART – EA based method

Step	SC	BC	CC	EC	TC	FC
1	99.2	97.3	98.6	85.2	100.0	76.86
2	100.0	98.9	98.6	92.6	100.0	91.16
3	**100.0**	**98.9**	**98.6**	**94.5**	**100.0**	**91.43**

A setup similar to that used for the PIA was implemented to generate the test sets for the UART: the constraints library counts about 440 lines of XML; μ was set to 30 and λ to 40, and the steady state parameter was, as above, 10 generations. It is important to note that the values of the evolutionary parameters in both experiments were selected to obtain quite similar generation times. Increasing the values of the evolutionary parameters would result in more accurate test blocks, but at the cost of longer generation times.

The run resulted in 3 test blocks. The total size of the generated programs is about 1,380 bytes, and the data set contains 231 input data. The time required to apply the test set is 101,685 clock cycles.

Setup time comprises the activities of analysis of the SoC, setting up the work environment and, for the EA-based methodology, writing of the constraints library and some support scripts, whereas generation time includes generation and simulation of test blocks.

Table 6. PIA – Timing figures for both generation methods

Times	Manual	EA-based
Setup [h]	30	54
Generation [h]	30	2
Total [h]	60	56

Table 7. UART – Timing facts for both generation methods

Times	Manual	EA-based
Setup [h]	30	60
Generation [h]	60	1.7
Total [h]	90	61.5

Table 6 and 7 clearly show that the total time for EA-based test generation is lower than for the manual methodology. A longer setup time is in fact more than offset by the shorter generation.

5. CONCLUSIONS

This paper presented an evolutionary approach to the generation of test sets for peripheral modules in SoCs driven by high-level Code Coverage Metrics (CCMs). The employed methodology allows direct comparison with an existing manual workflow, and compares favorably against it.

The EA-based method achieves better results in terms of CCMs and fault coverage, and is cheaper than manual method in terms of memory occupation and generation time. An EA, able to manage flexible individual representations and to concurrently generate mixed but highly correlated individuals, has been devised and is also described.

For the purpose of evaluating the relation among RT- and gate-level metrics, during the development of the PIA's and UART's test sets the authors performed some gate-level fault simulations. These simulations have confirmed that increasing RT-level code coverage values always mean increasing gate-level fault coverage values, and to conclude that a correlation between these two groups of metrics exists.

Currently, the authors are working on an improvement of the presented evolutionary method in order to reduce the required setup time.

6. ACKNOWLEDGMENTS

The authors thank Alessandro Aimo, Luca Motta and Alessandro Salomone for their invaluable help in designing the µGP[3], and Alberto Cerato for performing most of the experiments.

7. REFERENCES

[1] N. Kranitis, A. Paschalis, D. Gizopoulos, G. Xenoulis, "Software-based self-testing of embedded processors", IEEE Transactions on Computers, Vol 54, issue 4, pp 461 – 475, April 2005.

[2] A. Manzone, P. Bernardi, M. Grosso, M. Rebaudengo, E. Sanchez, M. Sonza Reorda, "Integrating BIST techniques for on-line SoC testing," IOLTS 2005: IEEE International On-line Testing Symposium, 2005, pp. 235-240

[3] S.Thatte, J.Abraham, "Test Generation for Microprocessors", IEEE Transactions on Computers, Vol. C-29, pp 429-441, 1980

[4] C.A. Papachristou, F. Martin, M. Nourani, "Microprocessor Based Testing for Core-Based System on Chip", ACM/IEEE Design Automation Conference, pp 586-591, 1999

[5] F. Corno, M. Sonza Reorda, G. Squillero, M. Violante, "On the Test of Microprocessor IP Cores", IEEE Design, Automation & Test in Europe, pp 209-213, 2001

[6] A. Cheng, A. Parashkevov, and C.C. Lim, "A Software Test Program Generator for Verifying System-on-Chip", 10th IEEE International High Level Design Validation and Test Workshop 2005 (HLDVT'05), Napa Valley, California USA: IEEE Computer, 2005, pp. 79-86

[7] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, "An RT-level Fault Model with High Gate Level Correlation", HLDVT2000: IEEE International High Level Design Validation and Test Workshop, The Claremont Resort & Spa, Berkeley, California, November 8-10 2000

[8] P. C. Maxwell, R. C. Aitken, V. Johansen, I. Chiang, "The effect of different test sets on quality level prediction: When is 80% better than 90%?", IEEE proc. of the International Test Conference, Oct. 1991, pp. 358-364

[9] Jimmy Liu Chien-Nan, Chang Chen-Yi, Jou Jing-Yang, Lai Ming-Chih, Juan Hsing-Ming, "A novel approach for functional coverage measurement in HDL Circuits and Systems", ISCAS2000: The 2000 IEEE International Symposium on Circuits and Systems, pp 217-220, 2000

[10] E. Sanchez, M. Sonza Reorda and G. Squillero, "Test Program Generation From High-level Microprocessor Descriptions", Test and validation of hardware/software systems starting from system-level descriptions, Ed. M. Sonza Reorda, M. Violante, Z. Peng, Springer publisher, 179 p, ISBN: 1-85233-899-7, pp. 83-106, Dec. 2004

[11] K. Jayaraman, V. M. Vedula and J. A. Abraham, "Native Mode Functional Self-test Generation for System-on-Chip", IEEE International Symposium on Quality Electronic Design (ISQED'02), pp. XX-XX, 2002

[12] E. Sanchez, L. Veiras Bolzani, M. Sonza Reorda, "On Test Program Generation for Peripheral Components in a SoC Resorting to High-Level Metrics", [accepted for publication on] 8th IEEE Latin-American Test Workshop, LATW2007.

[13] R. Chandramouli and S. Pateras, "Testing Systems on a Chip," IEEE Spectrum, Nov. 1996, pp. 1081-1093.

[14] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, "Efficient Machine-Code Test-Program Induction", CEC2002: IEEE Congress on Evolutionary Computation, Honolulu, Hawaii (USA), pp. 1486-1491

[15] F. Corno, E. Sanchez, G. Squillero, "Evolving Assembly Programs: How Games Help Microprocessor Validation", IEEE Transactions on Evolutionary Computation, Special Issue on Evolutionary Computation and Games, Dec. 2005, vol. 9, pp. 695-706

[16] http://aspspider.org/alexsalo/

[17] http://www.opencores.org