POLITECNICO DI TORINO

SCUOLA DI DOTTORATO
Dottorato in Ingegneria Informatica e dei Sistemi – XX ciclo

Tesi di Dottorato

Design techniques for high-reliability complex electronic systems

Massimiliano Schillaci

Tutore
prof. Matteo Sonza Reorda

Coordinatore del corso di dottorato
prof. Pietro Laface

Marzo 2008

Summary

As the market continuously requires ever more powerful systems, there is a huge economic incentive in technological advances that can make that required power available to the end user of the system. Conversely, more powerful devices make it possible to integrate a large number of functions inside a single system. Eventually, users want to assign mission-critical or safety-critical tasks to digital systems.

This kind of delegation makes reliability a major requirement for digital systems. The activities to achieve system reliability include validation, verification, testing of the single components and of the entire system, diagnosis of the components, system hardening. These activities in turn benefit from the development of methodologies and tools for their effective application.

The performed activity concentrates on test and diagnosis of digital programmable circuits, with particular emphasis on microprocessor cores and micrcontrollers. In particular, the focus of activity has been the development of approximate methodologies for circuit test and diagnosis, and the assessment of those methodologies on realistic case studies. The activity also touched the field of system hardening, with the development of a methodology having the property of being transparent.

As a support activity for the test and diagnosis field, work has been performed in the field of evolutionary computation. The work includes the development of evolutionary methodologies, the definition of tool architecture, the coding, total or partial, of such tools, and the application of evolutionary approaches to problems seemingly far from the CAD field, with the purpose of enhancing the tools performance.

The thesis is structured as follows. Chapter 1 provides a very brief introduction to the fields of test, diagnosis, system hardening and evolutionary computation, also defining the terminology for the following. In Chapter 2 the activities performed in the field of test are presented. In chapter 3 the activities pertaining to diagnosis are described. Chapter 4 details the activity performed for system hardening. Finally, chapter 5 describes the diverse activities performed in the field of evolutionary computation.

Contents

Chapter 1

Introduction

The market increasingly requires complex systems, able to perform multiple functions and satisfy user expectations in terms of performance, usability, safety, security and so on. This need is satisfied thanks to the advancements in technology and design methodologies, that make the complexity of digital circuits increase constantly over time. The increased complexity of digital circuits makes it possible, and also desirable, to concentrate even more functions in a single system or device, in a positive feedback loop. Users find themselves relying on digital systems for more and more leisure and work activities, and critical applications are delegated to digital systems. The sheer amount of delegation makes reliability of digital devices and systems a crucial requirement.

Digital programmable systems, in particular microprocessors and microcontrollers, provide the computational power needed for overall system operation. The use of programmable systems allows to move system complexity from hardware to software, with several advantages: total cost can be reduced with a proper partition; the greater flexibility of software with respect to hardware makes system upgrade easier and cheaper; programmable components, even specialized ones such as DSPs and peripherals, can be reused over different systems, leading to further reduced component and design costs.

The activities to achieve system reliability include validation, verification, testing of the single components and of the entire system, diagnosis of the components, system hardening. These activities in turn benefit from the development of methodologies and tools for their effective application.

The purposes of these activities are different. Validation is necessary to ensure that the requirements for a system reflect its intended purpose; since it is concerned with a human idea of what the system should do, it is very difficult to automate. The purpose of verification is checking that a given design of a system respects its requirements. Test demonstrates whether a physical implementation of the system is defective or not. The locaton of a defect in a faulty device is pinpointed through diagnosis. Finally, hardening is performed to reduce the probability that a given system stops working properly after deployment.

The activity focuses on the testing and diagnosis of digital circuits, with particular emphasis on microprocessor cores. In particular the field of software based self test is

investigated, taking into account its possible applications as a cheap part of a complete end-of-line test campaign, for on-line testing of digital systems, and for incoming inspection activities. The software based diagnosis is also taken into account, developing specific methodologies. As reliability benefits not only from fault discovery but also on fault prevention and fault tolerance, techniques for hardening of digital systems are developed and characterized.

As a support activity for the automation of test set generation, the development and use of tools for evolutionary computation is undertaken. Development comprises the choice of a theoretical model and definition of the general architecture for several tools, and at least part of the coding. The tools, existing or newly developed, are then evaluated against CAD applications, which constitute their main task. For better theoretical and empirical understanding, and to improve their general performance, they are also proven in more general environments, such as games.

1.1 Test Basics

As microprocessor and microcontroller technology is more widely used day after day, it is essential for the manufacturers to provide well-functioning and reliable devices to customers and end users. The most direct way to achieve this goal is through *test* of the circuits [1]. Test is the activity through which correctly functioning devices are distinguished from malfunctioning ones. Circuits belonging to the first group are called *good* devices, and the others are the *faulty* ones [2]. A manufacturer will then sell only the good circuits, and discard the others. A part of the faulty devices, however, will normally be subject to further analysis to discover the causes of failure.

Actually the activity is performed trying to demonstrate that a particular device is faulty, by feeding it with inputs and comparing its outputs with a set of expected values. If the actual outputs are all equal to the expected outputs then the circuit is labeled good, otherwise it is faulty. If the circuit is labeled good the test is *passed*. From the above description follows that a circuit can be demonstrated faulty, but confidence on the correct functioning of a device cannot be complete.

The physical mechanisms by which a circuit may fail are uncountable, but to undertake a quantitative approach to the test problem some simplification is necessary. To account for the possible failure mechanisms several *fault models* have been introduced in the literature. A fault model is an abstraction of the failure mechanism of a circuit. The use of a fault model allows limiting the possibly infinite variety of physical defects and failures to a finite set of logical faults belonging to the model. In this way a fault model provides a *metric* with which the effectiveness of a test can be assessed.

Given a circuit containing a single fault belonging to a model, if a test detects a difference between the faulty machine and the good one then the fault is *covered* by the test. If, for a given fault, no test can exist that covers it, then the fault is *untestable*. Given a circuit and a fault model, a set of possible faults for the circuit compose a *fault list*. If the list contains all possible faults for the circuit, it is *complete*. A *test set* is a collection of single tests for a circuit. It should be noted that the definition of test set is independent

of the fault model. Its measured effectiveness may change with different models, but its definition remains intact, since it only includes a set of inputs and expected outputs. The ratio between the number of faults covered by a given test for a circuit and the number of faults in the complete fault list is the *fault coverage* for the test.

The most widely used fault model is the *single line stuck-at* fault. The hypothesis is that in a faulty circuit a single input or output pin of a logical gate is permanently fixed at a single logic value, thus modifying the logic function of the circuit. This fault model has been introduced in the early '50s, and now suffers from several drawbacks. It is not physical, since there is little relation between the fault model and the actual failure mechanisms, and it actually captures only a small class of real defects. It may be too simplistic, because it postulates the presence of at most one fault within a circuit, even in the case of very large devices. Most importantly, it does not take into account any timing effects, as it is a static model. As the operation frequency increases the defects linked with excessive delay and generally with incorrect timing within a circuit become more and more important. This last drawback is the most serious, and it is the main driving force behind the development of different fault models.
Even with these limitations, the stuck-at is still the most universally used fault model, for several reasons. It is simple, allowing fault simulation even for large circuits. Its cardinality, the number of possible faults for a given circuit, is limited, allowing simple management of complete fault lists for most circuits. It provides a metric of fault coverage. Most importantly of all, it has been empirically demonstrated that an insufficient fault coverage on the stuck-at fault model leads to unsatisfactory defect levels at the end of the manufacturing line. Thus, coverage of the stuck-at faults may not be enough to guarantee product quality, but it has been proved necessary.

Other important fault models are the *transition delay* fault and the *path delay* fault. These account for timing effects in circuits. A transition delay fault exists when a single transition on an input or output pin of a gate is propagated with a delay too high with respect to the nominal one. The effect of this fault may be the capture of incorrect values on the outputs or the memory elements of a circuit. The transition delay fault model is still relatively simple, since every fault is concentrated on a single circuit pin, and its cardinality is the same of the stuck-at fault model. The path delay model is similar to the transition delay model, but faults affect transitions along an entire *path*, a connected sequence of logic gates.

All the above models describe *permanent* faults, that is faults that always exist in the machine. As technology allows shrinking the devices, the associated electric charge decreases. This makes the device sensitive to the action of charged particles, such as cosmic rays and the products of radioactive decay, ionizing radiation and electromagnetic noise. Even if these do not permanently damage the device, they may change the voltage levels of one or more internal nodes in the circuit and lead to incorrect computation results. Malfunctions that do not modify the logic function of the circuit, but only temporarily change the logic values are modeled by *transient* faults.

If a transient fault exists in the circuit it might not pass a test. In this case the circuit is faulty, but only at the time of test, so there is the risk that it is discarded incorrectly. For this reason in some production environments a faulty circuit is tested again, and only

discarded if it fails twice.

Two important transient fault models are the *single event upset* (SEU) and the *single event transition* (SET). They both model the effect of a particle or ionizing radiation hit on the circuit. A SEU is the flip of single bit inside a memory element of the circuit, in a given instant. This change can lead to incorrect result as the content of the memory element is used as input for further logic gates. The SET is a natural extension of the SEU model, and is relative to all nodes in the circuit, not only memory elements. The SET can model interferences on long interconnection lines, or on circuit nodes driven by very small transistors.

From the above discussion follows that it is important to test digital devices *at-speed*, that is at their nominal work frequency. Indeed, many defects in the machine cause incorrect signal timing, and only show up when the circuit is used at its maximum frequency. This is true even if the test has been generated targeting only a static fault model. The incorrect results produced by a dynamic fault may be detected by such tests, but if the test is performed too slowly, defective logic gates may have time to drive signals to their correct logic value, and the timing error would not be apparent.

Despite the efforts spent by the research community over decades, testing of digital systems is far from a closed field. This is most apparently testified by the fact that no single test methodology is able to satisfy all the needs of the different users. Different methodologies achieve different balances between effectiveness, cost, speed, data size. Furthermore, as technology advances different physical mechanisms become responsible for malfunctions in a digital system, and this change is reflected in the models of faulty systems.

Not all methodologies are applicable by all users, as they have different levels of information available about the system tested. Different users also have different testing needs.

The circuit manufacturer has the highest level of information, since it knows all the details of the device. It usually also has access to test facilities purposely built into the circuit. On the other hand, the manufacturer needs to comprehensively test its device, since very low defect levels must be guaranteed to the customer. Because of the production volumes, however, test should take as little time as possible, to avoid the related additional costs. Adding to the problem is the fact that the automated test equipment (ATE) used at the end of a manufacturing line may cost millions of dollars each, and every second of test can mean several dollars more in the production costs of a circuit. Since production yields are usually not very high, to minimize costs manufacturers usually test their devices before packaging them, and then a second time after that.

But the manufacturing process is not perfect, and by the time a customer assembles the devices into its system, less than ideal transport and environmental condition may have introduced new defects in the circuit, or worsened others that were too small to cause failure. The customer, then, usually wants to test the device again to ensure proper functioning, an activity called *incoming inspection*. The level of available information is now lower, since the manufacturers protect the details of their circuits and technology as trade secrets. Access to special test facilities is also impossible or reduced.

In the following the focus will be on the test of programmable digital devices, with particular emphasis on microprocessor cores and microcontrollers. Development of different

test methodologies is the subject of chapter 2.

Processor cores are among the most complex circuits used today, and can be tested using several different methodologies. Test application techniques for microprocessors are traditionally divided in hardware-based and software-based methodologies. The former are directly derived from the traditional test tehniques for combinational and sequential, not necessarily programmable, circuits. Hardware-based techniques postulate the use of sophisticated ATE, able to impose the correct timing on the input signals and collect the outputs measuring accurately their delay. The ATE is also in charge of providing the clock signals, and possibly to skew the input signals within the nominal tolerances, to check that the circuit works in all conditions matching its specifications. The cost for an ATE able to test a high-end microprocessor at its nominal speed is very high.

Hardware-based techniques usually include the insertion of special hardware to ease test application and improve fault coverage. The memory elements in the circuit are linked together in one or more scan chains, that allow setting the values of at least some registers and reading them afterwards. The memory elements of the circuit then become additional controllable and observable nodes. The addition of scan chains to a circuit impacts both area and delay, which translates to higher design costs and perhaps reduced yield. Other scan structures containing the memory elements of a circuit are possible, such as *level-sensitive scan design* (LSSD), first used by IBM, or other schemes. Hardware-based methodologies are generally very effective, leading to high fault coverages, at least for the stuck-at fault model. On the other hand, they can be very costly in terms of design effort, area overhead and performance penalty. Such techniques also require the use of very sophisticated ATE, since the test equipment is in charge of managing all the timing. In addition, hardware-based techniques are effectively useable only by the circuit manufacturer, since they require both detailed information about the internal structure of the circuit and free access to the additional test structures. Scan chains are not readily useable by the customer of a processor core, since he lacks essential information to use them.

One of the difficulties with the use of scan chains is that the test generated to use them can cause a power consumption that is even an order of magnitude higher than nominal values. This happens because the scan chains allow putting the machine in a state that would never be reached during normal operation, and has therefore not been catered for in power computations. This excessive power consumption leads to overheating the device (or, worse, parts of it), and may damage it. This problem is particularly vexing in very large circuits. On the one hand, the scale of the circuit forbids a large overestimation of the current drain, because that would lead to excessive costs in cooling systems for the finished product. On the other hand, the use of test vectors specially generated to keep the power consumption nearer to the nominal values can lengthen by far the test application time, again impacting costs.

In contrast, software-based techniques, named *software-based self test* (SBST) techniques, rely on the key idea of exploiting the computational power of the tested core to apply test patters and collect the results. The technique, pioneered by Thatte and Abraham [3], has been subsequently developed by numerous researchers [4] [7] [8]. The processor under test is used in normal mode, without the need to resort to a special test

mode. The test is performed running a set of programs whose purpose is not to perform useful computations, but to extract information about the microprocessor. Test programs excercise the functional modules of the processor, collect the result and make them available for external observation.

The advantages of SBST are several. The circuit is tested *at speed* since it is used in normal mode. It avoids problems of excessive power consumption. Since there is no need for strict external control of signal timing, lower cost ATE can be employed. Being a software methodology, SBST is relatively technology-independent, meaning that technology changes have a much lower impact on SBST than they have on hardware methodologies.

The attendant disadvantages are that SBST, providing less control and observation power than scan-based techniques, generally has lower fault coverage capabilities, and writing good test programs is a challenging task.

Several different metodologies have been proposed in the past to generate test programs for processors. The classic approach [3] is based on the modeling of the processor as a directed graph. In this graph the vertices correspond to registers inside the processor or to memory systems outside the processor, and the edges represent the possible data transfers between these memory elements. Edges may be labeled with additional information, describing which instructions are responsible for the transfers. The test checks all the possible transfers between the memory elements. It is theoretically very powerful, since it takes into account many different parts of the processor, but also has some drawbacks. First, it leads to very long test procedures, since the graph describing the processor may be very big, especially regarding the number of edges. A long test procedure then translates to a long application time. Additionally, it does not take into account several structures that have become of widespread use after the methodology was devised, such as the pipeline or the speculative execution blocks (branch predictor, secondary pipelines, ...). The pipeline registers may be included in the model, further increasing the test length and size, but other blocks cannot, and require different methodologies.

A newer approach [4] employs compact routines, each targeting a specific processor module, obtaining a high fault coverage for that block. Blocks targeted in this way are the functional units of the datapath: the multiplier, adder, shifter and arithmetic-logic unit (ALU) are specifically tested. The purpose of each routine is to generate a set of test vectors for the target module, apply them and collect the results. This approach is generally effective on the datapath modules, but it has its own shortcomings. The control part of the processor is not explicitly taken into account, and is tested only indirectly, obtaining less satisfying and less predictable coverages. Control logic for a processor includes not only the instruction sequencing blocks, but also the pipeline control logic and the branch unit, where present.

A recent extention of the above approach targets the floating-point unit of a processor [6]. The advantages and attendant drawbacks of the methodology are similar. The target modules are extensively tested, but the control part is not explicitly exercised.

Another methodology, named *DEFUSE*, [7] has been proposed for generating test programs for microprocessors, aimed at the arithmetic modules. Again, the control parts are not directly tested, and their coverage is not optimal.

A different, and somewhat interesting, methodology resorts to the so-called *FRITS* kernels [8]. These are programs that repeatedly generate and execute pseudorandom code fragments. To allow the application of the methodology at the end of the manufacturing, when the processor is not yet linked to a system, all the code is loaded into the processor's cache, and executed within it. To be able to use low-cost ATE, with low pin count, special care has to be taken not to generate any external memory access. The methodology is used by the industry, and it is reported to provide valuable additional coverage with respect to other test techniques. However, it requires a deep knowledge of the processor under test to optimally tune its parameters. It also puts some design constraints on the manufacturer, since the processor must be able to load the test into the cache and then restart without invalidating it. It is therefore a valuable methodology for the producer, but may not be applicable by the user of the device.

Apparently, no single test generation or application methodology is able to satisfy the needs of all users. Even in the field of test program generation several approaches achieve different balances between complexity, effectiveness and widespread applicability.

Often the customer is usually only interested in testing the device for its own special application. It also usually has lower manufacturing volumes than a circuit manufacturer, so longer test times are affordable. Some defects only show up in environmental conditions that are very different from those at the end of the production line, but may exist during the field life, and are known by the customer of the device. Finally, the ratio of working devices (the production yield as perceived by the customer) is high, so it makes sense to skip a preliminary test phase and perform incoming inspection activities after the system is assembled, by testing it thoroughly once.

The longer test times available and the application-specific nature of the system produced make it possible to undertake a stress test activity on it. The term stress test here does not pertain only to environmental conditions, but also to the level of load induced on the system and to the coverage of all possible working conditions.

1.2 Diagnosis Basics

Digital circuit technology suffers historically from low production yields. This is especially true for high-end devices, which push the envelope of design procedures and technological processes. The market, in fact, shows an almost insatiable request for faster computation systems, and the manufacturers answer this need with aggressive design rules, intended to get the most from the available technology. These design practices, however, also highlight the existing criticalities in the manufacturing process, resulting in a large percentage of failing devices. In the high-end market, however, the customer is willing to pay premium prices for the most performing devices and systems, and procedures such as speed-binning can recover at least a part of the production for the lower-end market.

The situation is different, but not necessarily better, for low-cost circuits. In this case the strongest pressure is for cost reduction in the manufacturing phase. This is achieved packing as many devices as possible on the silicon wafer, and at the same time keeping yield as high as possible. These two last goals may be conflicting, as smaller

circuits usually mean smaller transistors and wires, and therefore a greater susceptibility to localized defects.

Once a digital device is found to be faulty it is usually discarded. But the activity may not end there. At least part of the failing circuits is analyzed to discover why a circuit has failed. The goal of this search is to understand what parts of the circuit or of the manufacturing process are most critical, and possibly correct them.

Manufacturers usually design and produce several successive releases of a given product. Different releases may feature new or improved functions, speed enhancements or logic design fixes. They also give the chance to modify critical parts of the design, in order to lower the impact of physical defects on the device. The two interconnected activities of fault localization and design update allow to increase production yield and to improve the reliability of the manufactured systems.

Before undertaking a lengthy analysis of the circuit to find the defect that caused its failure, it can be useful to limit as much as possible the area to explore. This can be done by finding a minimal set of logical faults that can have caused the failure. Although it is true that a logical fault does not necessarily describe a real physical failure mechanism, the location of the faults belonging to that minimal set is directly linked to the location of the actual fault.

The *diagnosis* is the activity through which, given a faulty device, a set of possible faults causing the failure is found. Diagnosis is performed through the application of *diagnosis sets* to the circuit. A diagnosis set is conceptually similar to a test set, but its purpose is to distinguish the faults, not only to find them. Test patterns in the diagnosis set must be accompanied not only by an expected response, but also with a criterion to classify the different faulty responses.

A test detects a fault if the outputs of the faulty machine are different from those of the good machine. Given two different faulty machines, if a test set obtains different results from the two machines than the test distinguishes the two faults. If, given two faults, no test set exists that can distinguish them, then the two faults are *structurally equivalent*. It is important to note that this property only depends on the circuit topology and on the fault model, it is not a property of the test sets. For a circuit it is generally possible to split the complete fault list in sets of structurally equivalent faults purely on the basis of a topological analysis.

Considering a different point of view, a diagnosis set splits the complete fault list into several sets for which it obtains the same results. The faults of each set are equivalent *with respect to the diagnosis set*. Every such set is an *equivalence class* (EC) for that diagnosis set [14]. For a given diagnosis set, an equivalent fault class is a subset of the fault universe including undistinguishable faults. Structurally equivalent faults will always belong to the same EC, no matter the considered diagnosis set. Every individual EC is completely disjoint from the others, and their union is the fault universe.

From the above definitions follows that every equivalence class must be composed by one or more sets of structurally equivalent faults. Given a diagnosis set, the set of equivalence classes relative to it define a partition of the complete fault set. No set of structurally equivalent faults can belong to more than one equivalence class, no two structurally equivalent faults can belong to two different equivalence classes.

The goal of diagnosis, therefore, is to split the complete fault set into the greatest possible number of equivalence classes. Another way to express this is that diagnosis aims at finding a partition into equivalence classes with the minimum possible average size.

Since it is already known that structurally equivalent faults cannot be distinguished by diagnosis the reduced fault list is customarily used. In the reduced list every set of structurally equivalent faults is represented by one fault. Even more than in the fault coverage problem, the use of the reduced list avoids useless computation. Of course, using the reduced list also decreases the average size of the equivalence classes. In the ideal case the reduced list is split in classes composed by only one fault each.

A fault is said to be *uniquely diagnosed* if it is distinguished from every other fault. In general this is possible only when using the reduced fault list as the fault universe.

Diagnosis is performed by applying a set of tests to the circuit. Every test splits the fault universe in two or more subsets. Depending on the results of the tests applied until a given moment, the set of possible faults in a circuit is reduced to a smaller one. The following tests must be able to split this subset, otherwise it is useless to apply them. To systematically indicate the tests to apply at every point in the diagnostic process, and which subsets can be formed depending on the test results, a *diagnostic tree* is built [15]. Every level of the tree corresponds to one test, and the branches indicate the possible outcomes for that test. Every node of the tree is associated with the set of possible faults in the circuits, that is with the equivalence class isolated so far.

Together with a diagnostic tree a *fault dictionary* can be obtained. It describes all the equivalence classes, and for every class it reports the outcome of each test. This structure allows to apply all the tests in sequence and then statically look up the resulting equivalence class, while the tree is more suited for a dynamic diagnosis process in which tests can be skipped if not useful.

A diagnostic tree can be binary if it only uses pass/fail information or n-ary if more complex information, such as multiple outputs, is used. Given a circuit and a set of tests, an n-ary tree may split the fault universe in more equivalence classes or be more compact. However, sometimes a pass/fail information can be obtained using cheaper equipment than that necessary to gather the full output from a circuit. In fact, a small logic module may be added to a circuit, whose purpose is to collect the outputs from the circuit and compute a signature from them. Comparing the actual signature with the expected one gives the pass/fail result.

The diagnosis activity is strictly linked to test. Exactly like test, it can be performed using hardware-based techniques, including the use of scan chains, or through software-based methodologies. Traditionally, tests have been applied using external ATE. However, technological progress is pushing up the complexity and operating frequencies of low end microprocessor cores. It has become apparent that parametric testing alone is not sufficient to achieve the high quality goals required. Moreover, even though ATE effectiveness on applying parametric test is unquestionable, the costs for an ATE able to run at-speed functional tests are becoming prohibitive for manufacturers of moderate quantities of units. As a consequence, the test community is heading towards alternative solutions.

The main alternative available is software-based diagnosis (SBD). In SBD the diagnosis set consists in a set of assembly programs and does not rely on any special test point to force

values or observe behaviors during application. Several researchers have proposed software methodologies for diagnosis, such as in [12]. SBD has the same advantages as SBST. It can be performed using low-cost test equipment, since the only purpose of the ATE is to load the diagnosis programs and collect the results. It is naturally performed at-speed, as the system runs software in its normal working mode. It avoids power consumption beyond the specifications of the circuit, for the same reason. It can be used when hardware structures inserted for test are not available or not working.

SBD is generally less powerful than hardware-based techniques, since the control and observation points are many more in the latter case. Lastly, it needs detailed structural information. Indeed, while test can be performed without any structural information, relying only on the functional specification of the system (although coverage measurements cannot be obtained in this case), diagnosis can be performed only with it, since its point is exactly to discriminate the faults from each other. No diagnosis can exist without a structural fault model.

Even more than test set construction, diagnostic set construction is a time-consuming activity. Most of the past effort in diagnosis was directed towards combinational circuits, whereas sequential circuits received less attention. This is due to the widespread usage of scan chain methodologies for test, that effectively turn a sequential circuit in a combinational one.

Hard to test faults require a high computational effort for their coverage, but once detected they are usually easy to diagnose. Indeed, they are covered by a few, very specific tests, which are different from each other. Faults that are easy to test, on the other hand, may be difficult to discriminate from each other and require a special effort for diagnosis. This difficulty can lead to long diagnostic tests, with correspondingly long application times and high costs.

Several metrics exist to express the quality of a diagnosis set. The effectiveness of a diagnosis set is usually assessed using measurements related to diagnostically equivalent fault classes. A diagnosis set is most effective if it is able to split the fault universe in the biggest possible number of ECs as small as possible.

The ability of a diagnosis set can be measured by means of its *diagnostic power*, defined as the fraction of all faults completely distinguished from all other faults, that is, belonging to fault classes of size 1 or comprising only structurally equivalent classes. The *diagnostic power for limit k* $dp(k)$ is defined as the fraction of faults that are classified into equivalence classes of cardinality less than or equal to k by the used diagnosis set. The diagnostic power then corresponds to $dp(1)$. Another possible metric is the *diagnostic expectation*, that is a simple average of EC sizes [16]. Finally, the *diagnostic resolution* or dr can be used, defined as the fraction of all covered fault pairs that are distinguished.

Algorithms for fault diagnosis can be broadly divided into two classes: the first exploits the *effect-cause* dependency, while the second one traces the *cause-effect* principle.

Algorithms in the former class analyze the actual responses and try to determine which fault might have caused the observed failure effect. These techniques are also known as *dynamic fault diagnosis algorithms* and they do not precompute a fault response database, but trace backward from each primary output to determine the error propagation paths for possible fault candidates. Effect-cause diagnosis methods can be further classified as either

symbolic or *simulation based*. Symbolic methods operate by building an error equation, while simulation based algorithms perform simulations to verify that a candidate location is capable to explain the faulty behavior.

On the contrary, algorithms in the latter class preventively store in a compact format all the information required to locate the set of faults that may be the cause for a circuit faulty behaviour. Memorization structures suitably aiming at this purpose are called fault dictionaries and their creation first consists in selecting, within the circuit faulty responses gathered by the used diagnosis set, a minimal subset of information allowing fault diagnosis.

Several decisional processes have been proposed in the recent past, leading to different dictionary organizations. Many approaches have been proposed to compress the dictionary size resorting to encoding techniques exploiting the regularity of the different organizations.

The most used dictionary organizations store diagnostic data in suitable matrices, tables, lists and trees. The quality of a dictionary organization may be given by the diagnostic resolution they afford. A *full resolution* fault dictionary is organized in such a way that no diagnostic information is lost during its generation with respect to the diagnostic abilities of the diagnosis set.

Several techniques permit generating full resolution dictionaries. The tree-based solution allows preserving the fault universe diagnostic classification provided by the diagnosis set.

In chapter 3 the activity performed is described. That activity belongs to the field of software-based diagnosis, and uses evolutionary tools for generation of additional tests.

1.3 System hardening

Microprocessor-based systems are used in an ever increasing number of applications, including mission-critical or safety-critical ones. Dependability for such applications is essential, and should be guaranteed during the development of the system.

For this reasons, the adoption of techniques to guarantee such dependability is a primary concern in the CAD field. *Hardening* is one of the techniques that improves the reliability and dependability of a system. Hardening is the activity through which a system is made less susceptible to environmental conditions, reducing its probability to fail during operation.

Environmental conditions that may concern developers are temperature, moisture, mechanical vibrations, presence of chemically aggressive compounds, electromagnetic noise and radiations. Of these, temperature and radiation take special places, for different reasons. All the other conditions, in fact, can be countered by enclosing the system in a robust, waterproof, suspended metal case. Depending on the environment, anodized aluminium, stainless steel or special resin covering may be needed for chemical protection. These are all mechanical or electro-mechanical hardening techniques, and only marginally concern the logic design of the system. Temperature is a special condition because the system itself will produce heat during operation, and dissipation or even refrigeration systems may be needed. The design of digital devices in this case is more important, and

techniques to counter thermal effects are the subject of low-power design. It is, however, out of the scope of this discussion, and is not generally considered a hardening technique, although it does have effects on the long term reliability of digital devices.

Radiation is peculiar for a different reason. In some applications it may be pervasive, meaning that no protection system is completely able to stop it. Indeed, ionizing radiation is a common condition in aerospace applications, where the atmospheric shield is reduced or absent. However, in these applications space and weight are luxury items, as the amount of fuel (and therefore money) needed to send an object to orbit is many times its weight. Heavy, bulky radiation-resistant cases are thus not an option for satellite systems, and it must be assumed that radiation will hit the system. Radiation may also be caused by the decay of particular chemical elements, such as thorium. This case is significant because small quantities of thorium and other radioactive elements may be contained in the ceramic packages of electronic circuits, that provide a good protection against other disturbances.

The effect of radiation can be modeled through the use of transient faults, such as single event upset (SEU) and single event transient (SET).

Hardening techniques can be hardware-based or software-based. Hardware techniques are exemplified by the classic triple modular redundancy (TMR) approach. The basic concept is that of executing the same operation more than once, and then checking to see whether it was correctly executed. In TMR the operation is performed by three copies of the device, and a voter block outputs the result that has been produced by the majority, in this case by two or three devices. The voter block must be built with special care, as it cannot be redundant. It should at least signal error conditions, even if the correct data could be obtained.

Other techniques use two copies of the hardware, just checking for a mismatch. In this case, the operation is repeated. Still other techniques may use even more than three copies of the hardware, to cater for the possible onset of permanent faults in them.

Hardware techniques are generally effective, and only introduce a small performance penalty. The common drawback of hardware techniques is cost. Even in the simplest case, the hardware cost is more than doubled with respect to the base system, as special logic for checking and recovery has to be implemented. Power consumption is also affected by the hardware redundancy, as more devices have to be supplied with current. For some applications, such as solar powered systems, this may be a greater concern than the total energy spent for system operation, as time may not be critical.

Software-based methodologies are centered on the idea of modifying the software executed by the system to achieve fault detection or even fault tolerance, whereas the underlying hardware is left unmodified. software may be modified at the high level or at the assembly level. Two main approaches are used: instruction replication and control flow checking [29] [30] [31].

For instruction replication all the program variables are duplicated, as well as the data processing instructions. The operations are performed twice on the two copies of the variables, and then the results are checked for consistency. If there is a mismatch then an error is detected. In this case execution may be halted, or the single operation may be repeated.

The idea of control-flow checking is that of inserting suitable instructions to track the program's execution flow. In particular it is checked that the flow is as expected every time a basic block is left and another is entered.

Software-based methodologies are quite effective in increasing the dependability of the processor-based system, but they usually introduce large time overheads, that may reach one order of magnitude. The reasons for this large penalty are twofold. First, both instruction replication and control flow checking introduce many additional operations inside a program, that represent a large penalty by themselves. Second, if this operation is performed at the high level, the optimization options of the compiler must be disabled to avoid removal of the redundant instructions.

A third class of techniques for hardening processor-based systems is that of *hybrid* methodologies. These have been originally introduced as an alternative to hardware only watchdogs, in order to reduce the large time penalties of software-based methodologies. Hybrid techniques exploit the concepts of instruction duplication and control flow checking, as used in software methodologies, to introduce redundancy in the computation and to track the execution flow. To this end they add a suitable hardware device, named *monitor*, to the system. The purpose of the monitor is to check the consistency of the duplicated computations and the correctness of the control flow execution.

This approach reduces the speed penalty by delegating the checks to the monitor. This happens in two ways. First, the consistency checks are no more executed in the processor, but the monitor takes care of them, working in parallel with the execution of the hardened code. Second, only instructions that communicate the change of basic block to the monitor are inserted in the code for control flow checking, while the monitor is in charge of checking that the switch occurs correctly.

Software methodologies and hybrid techniques share some common limitations and downsides, that may preclude their use in some applications. Instruction duplication and control flow checking increase both the amount of data used by the program and the program size significantly. This directly translates to a large memory occupation, up to 7 times that of the original system. These large memory requirements may not be acceptable in some applications, for cost reasons or simply because the processor is not able to address all the needed memory. If the hardened system is derived from an already existing one, this may be a serious limit.

Both instruction duplication and control flow checking require access to the application's source code. Modification may be performed manually or using dedicated automatic translators, but source code availability is nevertheless mandatory. Many applications, or off-the-shelf components, cannot be obtained in source code, as their producer protects the copyright on the product. One notable example is the code for operating system modules or libraries. This limitation may be as serious as the previous one.

1.4 Evoutionary Computation Concepts

A research activity in the field of evolutionary computation (EC) has been undertaken as a support for the test and diagnosis activities. Its primary goal was to automate

the generation of test programs for microprocessors, providing a general methodology to complete existing test sets.

Although generally effective, manual methodologies for the generation of test sets suffer from several drawbacks. The first, and most obvious, is that it takes a highly skilled test engineer to devise a test set with high fault coverage. Although capable of covering critical, perhaps difficult, corner cases, humans are not always able to cater for all possible usages of the device. Furthermore, the very process of understanding the device may lead to some bias in the test generation activity.

Evolutionary methodologies do not suffer from these limitations. They are stochastic optimization processes, driven by a metric that expresses the quality of the attained solutions. An evolutionary algorithm (EA) is able to cover usage cases that were not foreseen by the test engineer. It does not necessarily possess any information about the inner working of the device to test, so it does not exhibit any bias that may be due to partial usage of that information. On the downside, an EA is seldom able to cover very peculiar corner cases.

Evolutionary methodologies may be used in cases where other automated methodologies could not be applied, or where it would be difficult to obtain results in the desired form. For instance, an EA can be used even if the detailed gate-level description of a microprocessor is not available, or if it would be too computationally expensive to use it. Functional metrics can then be used to generate a test set suitable for application by the user of the device, in activities such as incoming inspection. Even in case an ATPG can actually be used, it would still be very difficult to put the generated test vectors in the form of a test program. This would make reuse of the test set on successive implementations of the device impossible, while conversely software-based self test is relatively technology-independent.

Evolutionary computation is an optimization *meta-heuristic* that mimics the Darwinian paradigm of natural evolution. The Darwinian theory of evolution maintains that living beings can develop complex structures and successfully adapt to their environment through the two key activities of *reproduction with modification* and *natural selection*. EC applies these concepts in a simplified manner, with the artificial evolution of possible solutions to a problem.

EC borrows most of its terminology from biology, as it is heavily inspired by it. However, most of the times biological terms are not used with their exact meaning, albeit transferred to an artificial field, but with subtle differences (sometimes not so subtle).

Given a problem, such as the generation of test programs for microprocessors, the possible solutions to that problem are mapped to *individuals*. In the simplest scheme, at every step of the process a finite number of individuals are kept, grouped in a single *population*. Each individual has a defined *fitness*, that numerically describes its ability to solve the target problem. It can be said that the problem to be solved represents the environment that the individuals should adapt to.

Individuals in the population reproduce, generating other individuals. Reproduction, however, is not a simple copy of one individual into another, as this would not lead to any evolution. Individuals are evolved, and therefore transformed in different individuals, through the random application of *evolutionary operators*. Individuals may reproduce

through *mutation*, a random change in one or a few points, or *recombination*, the random merging of the characteristics of two (or, very seldom, more) individuals.

A population is generally defined as the set of individuals that can exchange genetic material, and that compete with each other. This term is used also in evolutionary approaches that do not use recombination, and therefore do not perform recombination. The population is not unlimited in size, so the individuals *compete* to survive inside it. Competition just means that the fitness of the individuals is compared, and the worst ones are discarded from the population. This mimics the natural phenomenon of competition between living beings for limited resources.

Competition can occur also for reproduction: usually the best individuals are selected preferentially to produce offspring, trying to exploit their good features for the next individuals.

Artificial evolution proceeds in discrete steps named *generations*. Every generation starts with a fully characterized population, that is a population for which the fitness of all individuals are known, as is any additional statistical information that may be needed. Then a reproduction phase is performed. New individuals are generated, usually starting preferentially from the best ones of the previous population, and evaluated, assigning them a fitness. Depending on the exact scheme, the new population may replace the old one or be merged with it. A *survival* phase then follows: based on the computed fitness, the worst individuals are removed from the population, returning it to the previous size.

Evolutionary computation is termed a meta-heuristic, meaning that it is not a single algorithm, applicable to a certain class of problems, but rather a scheme for the application of stochastic optimization techniques and transformations.

EC can be applied for all those problems where the structure of an optimum solution is not known exactly, or where the optimum itself is not known. It is generally suitable for all those situations where an exact solution to a problem cannot (or should not) be searched for. These include problems with multimodal fitness functions or large search spaces.

One of the key concepts of EC is that it is not *finalistic*, exactly as the Darwinian model of evolution. This means that it is applicable, in principle, to any class of problems, because it does not embody any bias that a targeted heuristic may contain. It is a stochastic optimization technique, so it does not guarantee that the optimum solution will ever be reached. It may, however, be used multiple times to increase the confidence that the obtained results are of good quality.

In the field of EC a distinction is made between the representation of an individual inside an evolutionary tool and its form as a solution of a specific problem. The two forms serve different purposes, as the internal representation should allow easy manipulation of the individual, whereas the second form often has to satisfy specific syntactic constraints, as is the case for an assembly program.

Copying biological technical terms, the internal representation of an individual is termed its *genotype*, and its usual form as a solution is its *phenotype*. Usually the genotype of an individual is the subject of explicit modification by the evolutionary tool, and the phenotype is evaluated to compute the fitness. In classical evolutionary algorithms, the phenotype is generated from the genotype, and never the other way round.

The genotype of an individual is divided in single *genes*, here defined as the indivisible units of information that pass from an individual to its offspring. A gene may be a single bit, a character, a numeric value, or even a very complex symbol, but are usually treated as monolithic units.

Each gene occupies a specific place in the genotypic representation of the individual. Every such place is a *locus* (pl. *loci*). An important difference between evolutionary approaches is whether the genotypes of individuals are composed of a fixed number of loci, or if they can increase. Clearly, theoretical analysis of the second kind of algorithm is much more difficult. On the other hand, some problems do not lend themselves easily to a fixed-genotype representation, as it is normal for different solutions to a problem to have different sizes. Again, assembly program generation is a perfect example: different programs usually differ in the number of instructions, and it is not simple to express all that variety in a rigid frame.

Each gene inside an individual assumes one of a range of possible values. Every possible value for a gene is an *allele* for that gene. One possible way to see the evolutionary process, described in its original biological meaning by Dawkins [35], is as a competition between alleles. In this view, all the alleles of a gene compete to occupy the locus. The measure of their success is their diffusion in the population, and the measure of their effectiveness is the differential fitness they give to the individuals that possess them. This depends on all the other genes, so if many individuals in a population undergo a similar modification, a successful allele may be supplanted by another one, previously not so valuable.

Evolutionary algorithms usually employ one of two different strategies to evolve good solutions to a problem. They are customarily called *plus strategy*, or $(\mu + \lambda)$, and *comma strategy*, or (μ,λ). In this notation μ refers to the size of the population at the beginning of a generation, and λ indicates the size of the offspring for every generation.

In the plus strategy the new generated individuals are evaluated, then the offspring is merged with the old population, that temporarily grows from size μ to size $\mu + \lambda$. After this merger, the population is sorted by fitness, and only the first μ individuals are retained for the next generation.

In the comma strategy, in contrast, the new individuals replace the old population entirely. The old individuals disappear, and the new population follows the same process as seen above: individuals are sorted, and the first μ are kept. To avoid losing the best solution found during the evolutionary process the best individual is saved in a separate storage if it enhances the best fitness obtained so far.

One trivial consequence of this scheme is that in comma strategy usually $\mu < \lambda$, while in plus strategy there may be any ratio between them. More interesting properties can be noticed, though, by observing that every individual can be seen as a point inside a *search space*. The search space for a given problem is the set of all possible solutions to that problem. Although it is not necessarily a space in the mathematical sense, it provides a metric of distance between two individuals.

With (μ,λ) strategy individuals are always modified, and the process avoids the presence of fixed points in the search space. In a wide sense, individuals execute an uninterrupted *random walk* in the search space. In contrast, with $(\mu + \lambda)$ strategy the best individuals found so far are always retained, and constitute the starting point for every

subsequent operation. The individuals in a plus strategy are clustered closer together than in a comma strategy.

In general the comma strategy is preferred when the fitness function is *deceptive*, meaning that it is easy to find a local optimum for it, but it may be very far from the global optimum. In contrast, the plus strategy is preferred when the structure of the solution is complex.

A third scheme is found in the literature, known as a *steady state* strategy. In this scheme new individuals are added one at a time in the population, immediately discarding the worst one. It is actually a special case of the plus strategy, in which λ is 1, and is also often referred to as a $(\mu + 1)$ strategy.

It should be noted, however, that often different evolutionary schemes have been developed by different research communities, and there is not a universal agreement on the terminology. So, for example, *evolutionary strategies* is a particular class of EA, the terms plus strategy and comma strategy are borrowed from *genetic programming*, another rather different class of EA, and the term steady state is sometimes applied to generically describe a plus strategy, opposed to the term *generational*, that refers to comma strategy. These examples should be enough to warn the reader: it is easy to find confusing and contradictory terminology in the literature.

An already existing, home-developed evolutionary tool, named μGP, has been used for test program generation and diagnosis program generation. It has subsequently been improved in various ways, and finally reimplemented from scratch. The improvement and reimplemetation steps benefited from the use of the tool in fields different from its original application, such as games.

The μGP is an evolutionary approach composed of three separated, but interconnected, blocks. They are an *evolutionary core*, a *fitness evaluator* and a *constraints library*, originally called *instruction library*.

The evolutionary core cultivates a population of indivduals, using recombination, mutation and search operators. It follows a variation of the $(\mu + \lambda)$ strategy, and features self-adaptation of several parameters, such as the operator activation probabilities, the operator strength and the tournament size. It also performs clone detection and fitness scaling, with optional extermination. It uses entropy and delta entropy to measure and maintain diversity in the population. Entropy is also used to implement an *entropy fitness hole*, again to maintain diversity. It can use multiple fitness values, either in a prioritized manner or as a true multiobjective optimizator. Finally, it supports parallel fitness evaluation by providing multiple individuals for evaluation and collecting the fitness values accordingly.

The fitness evaluator accepts one or more individuals as input, and computes a fitness for each of them. It often takes the form of a script that sets up and launches a simulation, collecting the results afterwards, transforming them in a form the evolutionary core can use. The use of existing, possibly commercial, simulators is the main reason why the evaluator is external. The external implementation of the evaluator, coupled with a simple interface, provides the maximum flexibility to the approach. The fitness function effectively defines the problem, and also the *semantics* of the individuals.

The constraints library has the purpose of limiting the possibly infinite variety of

productions from the evolutionary core, and at the same time of providing a mapping between internal and external representation, that is between genotype and phenotype. It contains a description of the allowed structure for the individual, and a description of the possible contents of that structure. This information is used both when an individual is generated, to check whether it is a *valid* solution to the problem, and in the mapping phase when the external representation of the individual is generated.

Full details on the tool, its different versions and various applications out of the CAD field are contained in chapter 5.

Chapter 2

Test Activity

Test is a basic activity for a device or system manufacturer. The basic theory of test has been laid out in the past, and currently the main contributions are to be expected from the development of new fault models. The industrial practice is an established part of the manufacturing process.

Still, the problem of testing digital circuits is far from solved. Technology advances allowed increasing the complexity of the circuits to the point where exact solutions could not be searched for. Different subjects need to test a given circuit, but they own different levels of information, so they need different methodologies to generate and evaluate a test. New uses of existing circuits may hamper the possibility of testing them thoroughly.

One of the biggest challenges of test is cost. This can be seen as the cost for applying the test, as well as the cost for generating it. As seen in section 1.1, software-based methodologies can ease the application costs, so an effort is needed in improving generation costs and effectiveness.

Most work in the test field has been devoted to the definition of approximate methodologies for test set generation, and to the improvement of the tools performance.

2.1 Test generation for pipelined processors

Generation of test programs has been performed for a medium-size microprocessor core. The goal of the activity is to enhance the performance of an automatic tool for test program generation. The purpose of the tool is to generate a set of test programs suitable for end-of-line test. These are generated as a completion of a hand-written set of programs.

To assess the effectiveness of each test program the processor core has been instrumented and implemented in a FPGA device. For performance reasons a simplified fault model is used, and coverages are reported relative to it.

The methodology has been developed on the LEON 2 microprocessor. This is a synthesizable core conforming to the IEEE-1754 specification, which formalizes the SPARCv8 architecture [19], equipped with hardware multiplier and divider circuits.

The core features a 5 stage pipeline, comprising *fetch*, *decode*, *execute*, *memory* and *writeback* stages. It is also equipped with separated instruction and data caches [18].

The description is written in VHDL and amounts to about 18,000 lines of code. The circuit has been synthesized with a commercial tool and targeted at a simple home-developed library, containing only simple logic gates as AND, OR, flip flops and latches. The resulting netlist includes about 65,000 gates, excluding the RAM modules needed to implement the caches.

The pipeline is described in about 3,000 lines of RT-level VHDL and its netlist is composed of approximately 37,000 gates, including 742 flip-flops.

The netlist has then been mapped to a Xilinx Virtex 2000E FPGA device, to speed up the activities of fault simulation.

Fault simulation is performed injecting one fault at a time in the model inside the FPGA and running the test programs. Although the fault simulation is serial, its speed is much higher than what could be attained in software, thanks to the speed of the FPGA.

The FPGA is hosted inside a PC, on a PCI board. This allows contolling the execution of the test programs from the PC. A management program effectively controls the processor clock, allowing monitoring of the processor activity on a clock by clock basis.

To perform the fault injection the circuit has been instrumented after synthesis: every flip flop is replaced by a scan cell that allows setting and reading its value at every clock cycle. In this way it is possible to inject permanent stuck-at faults and transient SEU faults in all the memory elements of the processor. In this case the memory elements constitute the architectural registers of the machine and the hidden registers between pipeline stages.

Because of the space limitations on the FPGA device it was not possible to instrument all the logic gates in the same way as the flip flops. The fault model targeted, therefore, is the single stuck-at on the pipeline registers only.

The programs are evaluated according to their coverage of the above fault model. The results of the test are stored in the architectural registers of the processor. The management software allows retrieving them at the end of the execution.

First, a set of test programs is written by a test engineer according to the methodology presented in [5]. Each of these programs targets a specific processor module. It executes a loop in which patterns are generated for the module and results are collected. Since the loops are long, the results are compacted in a signature, that is stored in the registers.

These programs are able to effectively test the processor's functional modules, and are especially targeted at the data path logic. In particular, the adder, multiplier, divider, shifter, and logic unit are exercised, obtaining a high fault coverage.

Other parts of the processor, however, are not effectively tested. In particular, the logic belonging to the pipeline is not targeted and the fault coverage on its registers is unsatisfactory, at 58.89%.

To add content to the test set and raise the fault coverage figures an automatic test program generation methodology has been employed. The μGPv2 tool has been used to evolve the additional test programs.

Actually, several different releases of the tool have been used, to be able to assess the effect of new features on the overall performance.

The reference version employs a plus $(\mu + \lambda)$ strategy, resulting in a totally elitist scheme. It uses two types of recombination and four random mutations as evolutionary operators. The selection scheme is a tournament selection with two contestants at a

time. For survival the first μ individuals in the ranked population are kept, discarding the following ones.

To this basic scheme three independent features have been added. The first one is a new evolutionary operator, named *local mutation*. It randomly selects a parameter inside a macro of an individual and changes it by a small amount with respect to the entire range. This operator has been introduced as a complement to the *random mutation*, that changes a randomly selected parameter to any value in the allowed range. The purpose of the local mutation is .

The purpose of the local mutation operator is to allow a random walk of the selected individual inside the search space, rather than a large movement, allowing an efficient searching of the nearby solution space around a high-fitness individual. This local search enables faster exploitation of the local maximums.

The second is the self-adaptation of the tournament size (τ) for tournament selection. In the basic version the tournament size is fixed to two participants, resulting in a selection probability equal to that of a linearized-fitness roulette wheel. Modification of the tournament size allows changing this probability distribution. An increased τ polarizes the distribution towards the best individuals, increasing the selective pressure. Decreasing τ, conversely, leads to a flatter distribution, so lower-ranking individuals see their chances for reproduction increase. The allowed range for τ is [1,∞]

Self-adaptation is based on the success rate of the evolutionary process in the last generation. Success in this case is measured as the generation of at least one offspring whose fitness is better than that of any other individual so far. If the best fitness has been improved, then

$$\tau_{new} = \alpha\tau + (1 - \alpha)\tau_{max}$$

where τ_{new} is the tournament size for the next generation, and τ_{max} is the maximum possible value for the tournament size. If, on the other hand, no new individual was better than its parents, then τ is decreased

$$\tau_{new} = \alpha\tau + (1 - \alpha)\tau_{min}$$

In these formulas, α is an inertia factor, set by the user in the range [0,1]. The inertia is fixed and never self-adapted.

The purpose of self-adaptation of the tournament size is to allow the tool to automatically choose between a hard and a soft selection model, based on the proceedings of the evolution. Selective pressure is increased when it seems easy to find a local maximum, while it is eased, using a softer selection scheme, when that is not the case.

Tournament size is increased when the tool is able to obtain significant fitness improvements and it is decreased when gains are small or nonexistent. In order to fine-tune the selective pressure, a continuous model has been applied to tournament size, so it can take on any real value between a minimum and a maximum. Expressing tournament size as $\tau = n + p$, where n is an integer value and p is a fractional number less than 1, then the actual tournament size is n with probability $1 - p$, and $n + 1$ with probability p.

The third new feature is an aging mechanism for the individuals. In the basic version, the individuals are removed from the population only in one case: when their fitness is

low enough to put them beyond the first μ individuals, or, better still, when at least μ individuals attain a better fitness. Individuals can be pushed off the population, but otherwise they are immortal.

The purpose of aging is to tune the behavior of the tool from a pure plus strategy to a comma strategy, that is to control the effects of elitism in the process.

A prototype of the proposed approach was built tackling such faults. The prototypical tool is composed of about 2,000 lines of C code for implementing the *fault manager* and the *automatic test program generator*. The hardware accelerator is driven by the *fault simulator* tool, composed of about 1,000 lines of C code.

Experiments were run on a PC equipped with a Pentium IV processor running at 1 GHz and with 1GiB of memory. Fault simulation and evaluation function computation were performed on the FPGA board. The whole process took about 26 hours to complete.

Table 2.1 summarizes the results: column "Fault Coverage" reports percentage of faults detected on the pipeline registers. "Clock Cycle" reports the number of clock cycles required to fully execute the test set.

The initial test set has been completed with automatically generated programs using the different releases of the tool. The attained fault coverage is raised from 59% to 100%, adding only 116 new instructions to the test set. The test application time is expanded only 1.4 times.

Test Set	**Fault Coverage**	**Clock Cycles**
Traditional	58.89%	11,263
Completed	100.00%	15,843

Table 2.1. Fault coverage results

To better evaluate the effectiveness of the proposed approach, a set of randomly generated test programs of comparable lengths were generated. Such a test set attains a fault coverage slightly above 78%, indicating that the considered faults are not easily testable.

Indeed, experiments showed that a random program containing 20,000 different instructions is not able to test all of them.

Five experiments have been performed to evaluate the effect of the proposed improvements to the evolutionary core. The first one has been performed using the previous version of the evolutionary core, and is called the Reference. Three experiments were made to isolate the effects of every single improvement to the tool, and are indicated by AGE, LOC and TAU. The final one has been performed implementing all of the improvements in the μGPv2 core, and is shown under the name Complete. The features employed in the five experiments are shown in table 2.2.

All of these experiments use the same evolutionary parameters: μ is set to 30 individuals, λ to 20 genetic operators. Evolution is performed for a maximum of 100 generations. For every experiment the random number generator has been set to the same initial value.

The results for all these experiments are in table 2.3. The rows refer to the experiments performed and to the result that could be expected if the effects of the modifications to the tool were linear.

Experiment	Elitism	τ self-adaptation	Local search
Reference	strong	no	no
AGE	relaxed	no	no
LOC	strong	no	yes
TAU	strong	yes	no
Complete	relaxed	yes	yes

Table 2.2. Experimental setup

Experiment	Fault coverage
Reference	97.8%
AGE	97.8%
LOC	97.7%
TAU	99.2%
Expected	99.1%
Complete	100.0%

Table 2.3. Fault coverage results for different improvements

All experiments start from nearly 97.5% fault coverage. The Reference then progresses little.

It could be expected that the LOC experiment would not perform significantly better than the Reference one because local search, although certainly useful to efficiently exploit the fitness function, brings with it the risk of becoming stuck in a local optimum. This is particularly true since a purely elitist scheme is implemented for this experiment.

In fact, it even performed worse than the original evolutionary process. This is a sure sign that the fitness function is a deceptive one. Looking closely at the experimental results it has been noted that the population is quickly filled by individuals that exhibit the same fitness value. The evolution process then gets stuck in a local optimum and the tool is not able to improve the population.

The AGE experiment greatly relaxed the elitist scheme, allowing greater freedom in the search space exploration. However, contrary to expectations, this did not immediately lead to improved results. The evolutionary process exactly followed the original one, so in this particular case the relaxation of elitism does not, by itself, produce any effect. It is noteworthy that in this experiment elitism has been relaxed, but not completely eliminated.

It is also reasonable to expect that tournament size self-adaptation, exploited in the TAU experiment, would give better results than a local search capability when used within an elitist scheme, since the rationale for this self-adaptation is exactly to avoid getting stuck in a local optimum. It gave better results than the simple elitism relaxation, since it was also able to increase the selective pressure when it was needed, enabling fast convergence of the evolutionary process.

The complete experiment outperforms all the other ones. This was expected, since several useful techniques were combined, composing their effects.

The differences obtained using every single tool update can be composed by summing the difference in fitness between each of the single feature experiments and the Reference one, obtaining an expected overall performance enhancement under the hypothesis of linear addition. The real achievements were better than linear; this confirms the correctness of the choice of the new features for the evolutionary tool. It can be noted that the combination of techniques that by themselves do not lead to enhanced results do indeed produce a sizable performance improvement.

2.2 Incoming inspection

An important test activity is incoming inspection, the activity through which the user of a device obtains the confidence that the device is working properly after delivery by the producer.

The user will employ the device in building an application specific system, so it usually needs to acquire such confidence before manufacturing, although tere may be exceptions. The reasons why a device may fail before assembly are various. Less than ideal transportation conditions, or system manufacturing environment, may introduce new defects in the device, or worsen the impact of others that did not previously cause failure. The device may be used in systems that have to work in peculiar environmental specifications, such as mechanical disturbances, electromagnetic interference, or thermal stress, that can cause failure in the assembled system. Lastly, the end-of-line test may not cover all the modes of operation required by the user. This last condition is uncommon, and generally only possible for very cheap systems.

The user of the device, in general, is also a manufacturer for other systems containing that device. These systems must satisfy end-user requirements of quality and reliability. To achieve these goals, the user must perform incoming inspection and system testing.

One user of digital electronic devices is Magneti Marelli, an automotive subsystem provider for several car brands. Magneti Marelli was born as a provider of magnets for electrical engines in 1919. Through time the firm began diversifying its products, manufacturing parts for submarines, trains, airplanes, and entering the consumer market. In subsequent years it retargeted its core business to the automotive market, and now provides engine control systems and body computers to several car makers.

Currently the average western car buyer puts several requirements on the car makers. A car should be powerful, energy-efficient, safe, comfortable and reliable. It should assist the driver in emergency maneuvers, signal mulfunctions, warn on potentially hazardous situations, handle adaptive active and passive safety systems, and provide leisure for passengers. The growing list of requirements makes the use of electronic systems mandatory for the automotive industry.

Also in seemingly mature fields, such as the *powertrain* control systems, market requirements drive the transition to ever more complex and powerful devices. Powertrain control system handle the operation of the engine, gearbox, differentials, turbochargers,

and generally all the mechanical components from the air and fuel intake to the axles on one side and to the exhaust on the other. In this case the requirements come not only directly from the users, but also from governments and authorities. In fact, due to environmental considerations, strict limits are put on the gaseous emissions that automobiles may produce per kilometer, both in terms of greenhouse gases and of other toxic pollutants. National laws may impose compliance to some emission limit to allow selling of the vehicles, directly pushing the producers towards the production of environmentally friendlier cars. Furthermore, both national and local regulations may deny circulation to vehicles not respecting the limits, this time putting the pressure on the user, who will in turn ask the car manufacturer for a better vehicle.

An engine control system should not only provide energy efficiency and low emissions. The car driver also expects power, a predictable torque, relatively low vibrations. All this contributes to comfort and mechanical reliability. A generic powertrain control system may also have to manage the gearbox, with either automated or semi-automated gear selection. The system should be efficient and provide fast and smooth shifts, again to balance performance and comfort. Additional functionalities for the control of turbochargers and differentials may be integrated in the engine control system or in more subsystems.

All these requirements contribute to the ever increasing usage of complex microprocessor and microcontroller cores inside engine and powertrain control systems. These are needed to provide the necessary computational power for the execution of all the intended functions.

Currently failures in electronic devices are becoming a significant source of breakdown in the automotive domain. Empirical evidence shows that a conventional screening approach aimed only at demonstrating overall system functionality is not effective in reaching the quality requirements posed by the users. Testing of the individual devices inside the system is therefore mandatory.

The activities of incoming inspection demand for environmental conditions more similar to those of in field use than end-of-line test. In addition, the production rates for automotive control systems allow longer test sessions to be carried out than is possible at the end of a semiconductor manufacturing line. Also, it is possible to stimulate the circuit with a workload similar to that expected during normal operation.

Automotive applications generally pose stricter requirements on electronic systems than usual consumer applications. The engine control system, indeed, lies inside the car engine compartment, and this implies a series of environmental stres conditions. First, temperature under the hood may easily exceed 100 °C, far above what devices commonly have to withstand. Electromagnetic noise exists in the vehicle, due to the presence of high voltage circuits driving the sparks, and large currents that power electrical engines. Mechanical vibrations are also present, as are potentially aggressive cheical compounds. For these reasons engine control systems are housed in waterproof metal cases for adequate protection. The case, however, cannot fully isolate the system from the environment, because it has to exchange signals with the external world. Indeed, the engine control system is directly responsible for at least part of the electrical spikes on the connection wires, as it drives or contains the fuel ignition circuits.

Some defects in digital circuits may only manifest in special environmental conditions,

not in normal applications. It is therefore essential to test the device in a condition similar to that of field operation. This operation is sometimes possible only during system test, as direct exposure of the device to the destination environment may damage or destroy it. As a consequence, activities that, strictly speaking, would belong to incoming inspection, are performed during system test.

A test program generation activity has been undertaken in collaboration with Magneti Marelli, with the goal of developing a methodology for an incoming inspection activity on the processor core inside a modern microcontroller. This is to be used in the latest generation of engine control systems for direct injection diesel engines (*MultiJet*$^{\text{TM}}$), compliant with Euro 5 regulations.

The user of a device does not normally have access to low-level information about the microcontroller. It cannot, therefore, use structural methodologies to generate the test. It can, however, employ a functional test. In this case software-based self test (SBST) is a privileged choice for test generation and application.

SBST methodologies rely on the computational power of the processor under test to extract logical information about its functioning. This information is then returned to the tester to discriminate faultu circuits from good ones. Logical information alone, however, may not be enough to discover faults.

Modern processors employ several mechanisms to improve performance or enhance reliability. The first kind of features include speculative blocks, such as branch predictors or auxiliary pipeline stages to perform computation that is not guaranteed to be needed later. These blocks improve performance by performing logical operations that are statistically useful in situations when, according to strictly sequential operation, the processor should remain inactive.

The second type of features is exemplified by frequency scaling in response to temperature changes. Since power consumption, and hence heat generation, is more than linearly dependent on operation frequency, a speed decrease can protect the device from overheating. Other techniques to limit heat include the selective switch off for logic blocks that are not currently used. Resuming operaton on these blocks, however, may not be immediate.

The common characteristics of these features is that they do not modify the logical result of computation, but they change the speed with which it is computed. A speculative module by definition may produce a wrong or useless result. This does not affect the outcome of the elaboration, but it just decreases performance. A fault in one of these modules, therefore, does not necessarily affect the numeric result of elaboration. A less than expected performance may be the only visible effect of such faults.

Similarly, thermal protection mechanisms may be used when it would not normally be appropriate. This may happen due to a fault in the module itself, or to faults elsewhere in the circuit that cause an excessive current consumption. In this second case the protection mechanisms work properly, and reduce the elaboration speed, but the performance decrease is not expected.

In all these cases a program takes significantly longer than expected for its execution. This erroneous timing points to an underlying defect, and is an essential measurement to detect it.

Detection of a critical timing is not only important for system reliability. Automotive applications such as engine control must operate correctly in real time. Severe damage to the vehicle and safety hazards may occur if the control system does not properly drive the power signals.

For these reasons a methodology has been employed, that performs a *stress test* on the microcontroller, through the collection not only of logical results, but also of timing measurements, to ensure proper functioning of the processor core.

In the case of microprocessors the stress test can take on several different forms, relating to the variable considered. The application of high temperature and/or humidity leads to a methodology called *burn-in* test: its main purpose is to eliminate parts whose reliability is too low for normal applications. The combined application of different supply voltages and clock periods is used to derive so-called *schmoo plots*, that define an area of logically correct operation for the device. On the other hand, the definition of a given workload is not trivial for microprocessor test, since its exact amount depends on the details of the microarchitecture. Control of a single variable does not allow to meaningfully apply a stress test.

For pipelined microprocessors, it is the sequence of instructions, and not only the single ones, that determines the behavior of the machine. The number of different instruction sequences is therefore a meaningful indicator of the usage level of the processor's pipeline. One hallmark of a stress test is that the processor is exercised for as long a period of time as possible.

In this case the stress test is implemented by a set of programs composed of two main subsets: *focused* and *non-focused* programs. Focused programs are those made to specifically stimulate one module of the architecture. They include programs that test the functional units of the processor, those that saturate internal resources such as a multiplier, and those that exhaustively apply all possible opcodes and addressing modes. The purpose of non-focused programs, instead, is to globally exercise the processor, making it execute very diverse code.

On the basis of the above points, the proposed methodology is composed of a mix of established SBST techniques. The first step tests the register transfers, based on the classic methodology described in [3]. Subsequently, the arithmetic and logical functional units are targeted, using an approach similar to [4]. Then the floating point unit is exercised with a methodology similar to [6]. Another test is performed on the floating point unit using the patterns from the classic approach for integer units. The pipeline is then targeted using a program manually developed by a test engineer. This program contains two main sections: one in which instructions are executed that have no data dependency from the previous ones, and one in which the dependency exists. This second part is further subdivided in several sections. In the first the dependent instructions are consecutive, in the others each couple of dependent instructions is separated by one or more independent instructions. The purpose of this methodology is to verify that all the forwarding mechanisms nominally available are actually active, and also that they are not used when there is no data dependency. After that, a test is performed on the branch unit: the processor state is purposely set, then a series of basic blocks is executed. These basic blocks are composed by a series of data processing instructions whose purpose is to compute a signature for the

test, ended by a single branch. To actually make the branches affect the sequential flow of the instructions the basic blocks are executed in a pseudorandom order. All this code is in turn executed inside a large loop. The initial processor state is chosen so that all branches are taken. This approach does not still ensure that the whole code does not result in an infinite loop. To enforce termination of the algorithm the branch condition codes are divided in classes, for instance the "greater or equal" class and the "less than or equal" class. Every class is associated with a register providing the state for the class. Before every branch, the state is updated in such a way that the number of taken branches for every class cannot increase. More formally, if a bit is assigned to every branch, where '1' means that a branch is taken and '0' means it is not taken, and the complete bit sequence is considered, then the processor state is updated so that the corresponding number starts as all ones and decreases monotonically over time. When the bit string is composed of all zeros, the basic blocks are executed sequentially. The state of the external loop is distinct from that of the branch classes. Finally, the pseudorandom test phase is undertaken. The program set incrementally computes an arithmetic signature composed by the content of all the registers. The global test information is composed by this signature and a measure of the time needed to execute the code.

The methodology described above is applied to an engine control system for a direct injection diesel engine (MultiJet^{TM}), compliant with Euro 5 regulation, and manufactured by Magneti Marelli. To match the computational needs the Freescale PPC5553 microcontroller is used [20] [21], running at 80MHz. The processor core is a 32bit, single-issue implementation of the PowerPC architecture, including a 7-stage pipeline, one integer unit for singleand multi-cycle operations, a vector unit also performing floating point computation, a branch processing unit with a branch target buffer (BTB), and 8kiB unified cache [18]. The controller also includes 1.5MiB flash memory and 64kiB static RAM. Additionally, both a time base and an externally clocked timing unit (eTPU) are available. The microcontroller implements the user level specification of the PowerPC architecture, extending it with a signal processing module and various interfaces.

The complete system runs a control software that occupies about 70% of the flash memory. Since it is part of an automotive system, the core runs a real-time environment.

The test is to be performed offline, during system manufacturing. As the manufacturing rate is in the range of 1 million units per year the total duration of the test has to be less than 6 minutes to perform all test activities, including subjecting the system to a predefined thermal profile and verifying power output and consumption. The time budget for functional test of the microcontroller is 10 seconds, about 3 of which are necessary to load the test program on the microcontroller and unload the results.

Since the core runs at 80MHz, this leaves about 500 million cycles available for test. Almost all tests are designed to incrementally compute a signature during execution. This is then provided back to the test machine through a serial connection. As time is a critical measure to determine the test outcome, it is measured in two independent ways, and then this information is added to the signature. The first measure is made using the time base registers, allowing to approximately count the clock cycles elapsed for the test. The second is is read from the eTPU; since this is clocked externally, it allows to measure wall-clock time. Both measures are necessary to reliably ensure correct execution. In

fact, the number of elapsed clock cycles indirectly measures the amount of computation performed, whereas actual time allows determining the speed at which this computation has been executed. Both measures should be within an expected range to conclude that the device is working correctly. In summary, a correctly working device has to produce the right logical value through the right amount of computation within the correct time.

To perform the functional test five main sections of code have been implemented. The first one tests the correct working of the general-purpose registers, and is derived from the original methodology detailed in [3]. Second, a series of short routines, taken from [4] and [6], is applied to test the integer and floating point units. Then a code fragment implements the test of the forwarding mechanisms. Since the execution module of the pipeline is implemented in three stages then the data dependent part of the code is split in three subsections: in the first the data dependent instructions are consecutive; in the second each couple is separated by one independent instruction; in the third they are separated by two independent instructions.

The approach has been implemented using instructions with different timings: single cycle instructions and three-cycle instructions. Actually the pipeline also contains a divider stage, but that is not pipelined, so it has been neglected. Subsequently a section of code containing a subset of all possible branches is executed. As described before, this section is composed of a series of basic blocks executed in a pseudorandom order. The initial state of the processor is chosen so that all branches are taken at least once. Not all possible branches are used for practical reasons: absolute branches can only have as target instructions within a small address range; the processor implements very complex and seldom used branch mechanisms, such as decrementing a count register, checking its equality with zero, branching to the address contained in the link register and saving the return address in the same link register. The use of such complex instructions is clearly limited. These four are focused code sections.

Finally, a non focused section is run. This is obtained using an instruction randomizer that obtains information about the processor ISA through an external library. In this way the user can target the test generation to different microprocessor cores, possibly from different families, and can also easily decide which opcodes to use during the nonfocused test and which to avoid. This is useful because many microprocessors provide instructions that require special care to use, such as cache management or multiprocessor synchronization. These instructions could hardly be expected to work at all inside random code, and are better handled in hand-written tests.

The randomizer used allows to make sure all opcodes in the library are used at least n times. The size of the code is predetermined to fit a memory budget, and the code generated by the instruction randomizer may not fill it. The remaining space is filled by pseudo-random code. This code can be seen as both focused and non-focused.

Indeed it extensively exercises the instruction decoder, implementing a nearly exhaustive test, but it also generally excites the processor's functions. The first four code sections have been written by hand, whereas the rest is automatically generated.

For the production test it has been decided to limit the occupied memory by code and data to 64kiB, but statistical results have been collected also for other allocated sizes.

Since no structural description of the microcontroller is available the results cannot be

expressed in terms of fault coverage, but only in terms of functional coverage.

The focused programs are characterized by the patterns they provide to a specific functional unit, and the expected fault coverage for that unit. Both the register test procedure and the short routines targeting the arithmetic unit are known to achieve a high fault coverage and very high functional coverage on their targets. The test program targeting the forwarding mechanism is designed to cover all possible forwarding paths, considering the fact that only three, out of the seven pipeline stages, can provide data to a subsequent instruction. Likewise, the branch testing routine is designed so that every branch instruction is executed at least two times, at least once as a taken branch and at least once as not taken. Actually, the majority of branch instructions are taken several times before switching to non-taken. Thus the targets for these instructions are stored in the branch target buffer (BTB). The storage of the branch address in the BTB and its subsequent removal when the branch is no more taken also affect the total execution time.

The target unit for an instruction randomizer is the instruction decode unit. Since all opcodes are used this amounts to a thorough testing. At the same time, executing the opcodes in a random order makes it possible to use the randomized opcodes as non-focused code.

In table 2.4 the application times for the five test sections are reported. It can be noticed that the non-focused code takes much longer (about three orders of magnitude) than focused code. As described above, not all instructions in the ISA have been used for non-focused code.

Target module	**Execution time**
Register	91.20μs
Datapath (ALU + FP)	899.02μs
Pipeline	39.03μs
Branch unit	49.95μs
Non-focused	2.994s

Table 2.4. Application times for the test sections

The instructions have been classified into several categories: Book E (those that a processor core must provide to be a compliant PowerPC architecture), cache locking, debug, Book E implementation standard (EIS), signal processing engine (SPE), scalar SPE floating point (SPFP), vector SPFP, variable length encoding (VLE) 16 bit, VLE 32 bit. Not all types of instructions are used during operation of the system, so they have not been included in the library. The excluded categories are SPE, vector SPFP and both VLE. This is important to note, because the excluded categories amount to a large number of opcodes.

Of the remaining opcodes those belonging to the cache locking, debug and EIS classes have not been used in the implemented tests, as they require specific programming sequences. Table 2.5 reports the coverage figures for the five instruction classes considered.

It can be noticed that not all the Book E opcodes have been used. These can be further divided in several subclasses to better gauge the properties of the non-focused test.

Inst. class	Used	Unused	Total	Coverage
Book E	150	54	204	73.53%
Cache l.	0	5	5	0.00%
Debug	0	1	1	0.00%
EIS	0	1	1	0.00%
Sc. SPFP	23	0	23	100.00%
Total	173	61	234	73.93%

Table 2.5. Opcodes covered for the considered instruction classes

Subclass	Used	Unused	Total	Coverage
Branch	1	11	12	8.33%
Cond. R.	11	0	11	100,00%
Inst. Syn.	0	1	1	0.00%
Integer	138	13	151	91.39%
PCRM	0	8	8	0.00%
St. Ctrl.	0	16	16	0.00%
Sys. Lin.	0	3	3	0.00%
Other	0	2	2	0.00%
Total	150	54	204	73.53%

Table 2.6. Covered opcodes for subclasses of the Book E instructions

Table 2.6 reports the details of the used instructions. In the first column Cond. R. denote instructions that operate on the condition registers, Inst. Syn. are those that perform instruction synchronization, PCRM stands for process control register manipulation, meaning instructions that operate on registers for process control, St. Ctrl. are instructions for storage control, and Sys. Lin. stands for system linkage. The greatest part of unused opcodes is clearly critical when used in random code. Branch instructions are hardly used, but this is not a problem because a focused test exist for the branch unit.

The purpose of the non-focused code is to cover as many working cases as possible on the processor core. For this reason it is meaningful to measure the number of different instruction sequences executed by the core. The PPC5553 features a seven-stage pipeline and is a single-issue machine, so instruction sequences are significant up to a maximum of seven instructions. In the following the coverage figures refer to the used opcodes, not to all the available ones.

The available memory space allows enforcing many times the execution of all opcodes. For instance, with 16kiB available the opcodes can be executed from 1 to 16 times before exceeding the memory budget. Varying the number n of executions changes the results in terms of coverage of instruction sequences. Considering the instruction couples, for small code sizes there is a definite coverage maximum for an intermediate value of n, whereas for larger code sizes the maximum coverage of instruction couples occurs for the largest

possible value of n. For the longest instruction sequences the opposite behavior can be seen: the maximum coverage occurs for the minimum value of n, that is 1. This different behavior is directly linked to the fraction of sequence space covered by the code: for a small coverage it is better to fill the space in a random way, whereas for larger coverages it is possible to increase the coverage by enforcing additional bounds.

Since the coverages are very low for sequences of more than four instructions, it is interesting to consider the maximum coverage that can possibly be obtained with every memory budget for instruction couples and triples.

Mem	**nOccur**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
16kiB	6	100%	9.7%	0.07%	4.4E-06	2.6E-08	1.5E-10	8.7E-13
32kiB	16	100%	16.9%	0.13%	8.6E-06	5.2E-08	3.0E-10	1.7E-12
64kiB	54	100%	28.1%	0.23%	1.6E-05	1.0E-07	6.0E-10	3.5E-12
128kiB	108	100%	45.3%	0.44%	3.0E-05	2.0E-07	1.2E-09	7.0E-12
256kiB	216	100%	64.7%	0.85%	5.8E-05	3.9E-07	2.4E-09	1.4E-11
512kiB	520	100%	78.6%	1.46%	1.0E-04	7.2E-07	4.6E-09	2.7E-11
1,0MiB	1040	100%	82.4%	2.73%	1.9E-04	1.4E-06	9.1E-09	5.4E-11
1,5MiB	1560	100%	82.6%	3.93%	2.8E-04	2.0E-06	1.4E-08	8.1E-11

Table 2.7. Coverage optimizing instruction couples

The various columns represent the coverages for single instructions (guaranteed to be 100%), couples, triples and so on, up to sequences of seven instructions.

The figures are low for the longest instruction sequences because the number of possible sequences is very large, on the order of 10^{18}. The reported data are consistent with what can be expected following a statistical analysis.

It can be noticed that the coverage for instruction couples is near 100%, and also that a saturation effect exists. Increasing the memory budget does not pay off for the coverage of couples as it does for other sequences.

The generation times for the non-focused code scale about linearly with memory size and tend to increase for large values of n.

Code size	**Time (n=1)**	**Max n**	**Time (n=max)**
16kiB	5s	16	4s
32kiB	9s	32	7s
64kiB	18s	65	17s
128kiB	40s	130	44s
256kiB	97s	260	124s
512kiB	230s	520	412s
1MiB	621s	1040	1,477s
1.5MiB	1167s	1560	3,097s

Table 2.8. Generation times for non-focused code

The total coverage figures for the complete test set has been computed taking into consideration not only the focused and the non-focused code, but also the initialization sequences necessary for correct execution of the programs. The overall figures are reported in the tables below. The final coverages are unevenly distributed among the considered subclasses: the scalar signal processing and floating point instructions are completely covered, whereas the instructions belonging to the Book E are partially covered. The instructions belonging to the other considered classes are not used at all.

Inst. class	**Used**	**Unused**	**Total**	**Coverage**
Book E	173	31	204	84.80%
Cache l.	0	5	5	0.00%
Debug	0	1	1	0.00%
EIS	0	1	1	0.00%
Sc. SPFP	23	0	23	100.00%
Total	196	38	234	83.76%

Table 2.9. Total coverages for the considered instruction classes

Again, a subdivision of the Book E opcodes in subclasses allows to better understand the characteristics and limitations of the methodology.

Branch instructions are used except for the absolute branches. This has a simple reason: the address field for absolute branches is a 16 bit wide signed integer, so they can only have as target an instruction in the first 32kiB of memory; the test routines, however, are not guaranteed to reside in a specific memory section. This forces the exclusive use of relative branches. The missing integer instructions deserve an explanation: actually, of the nine missing instructions two are traps, four are load/store instructions with reverse endianness, two are synchronized memory reads and writes, and one is an instruction that copies the exception register in the condition register.

Subclass	**Used**	**Unused**	**Total**	**Coverage**
Branch	8	4	12	66.67%
Cond. R.	11	0	11	100.00%
Inst. Syn.	1	0	1	100.00%
Integer	142	9	151	94.04%
PCRM	4	4	8	50.00%
St. Ctrl.	4	12	16	25.00%
Sys. Lin.	1	2	3	33.33%
Other	2	0	2	100.00%
Total	173	31	204	84.80%

Table 2.10. Total coverages for the subclasses of Book E instructions

2.3 Test of Peripherals

System-on-chip (SoC) and system-in-package (SiP) devices integrate several logic cores into a single silicon chip or into a single compact package. This integration, made possible by the advances in manufacturing technology, allows designing systems that are faster and consume less power than traditional designs. The SoC design paradigm, introduced in the '90s, makes it possible to keep the time to market low, by reusing already existing logic cores. On the downside, the tight integration of SoC components reduces accessibility to the internal nodes of the circuit, making test of the single logic blocks harder.

SoCs are generally composed by one or more processor cores, some memory cores and several peripheral cores, possibly in addition to application-specific logic modules. Peripheral cores in the SoC need to be tested before being put to service. Traditionally the activity of test generation for peripheral cores has not attracted very much the attention of the research community.
There are several reasons for this. Peripheral cores are often much simpler than other cores in the SoC, usually an order of magnitude smaller than processor cores. Being smaller, peripheral cores constitute less of a problem for test generation and fault simulation. Stand-alone peripherals are relatively easy to test, thanks to their structure, simpler than other circuits, and to their function. In fact, whereas a processor devotes most of its resources to computation of its internal state, materialized in registers, the main purpose of a peripheral is to transform inputs in a particular format into outputs with a different electrical or logical representation. Additionally, when methodologies based on hardware insertion are used, the resulting overhead in area and performance is often negligible.

However, the integration of peripheral cores in a SoC introduces new problems in the test of peripherals. First of all, the reduced accessibility of the module makes both test application and collection of the results harder. In addition, peripheral cores may be purchased as IP cores, and thus not modifiable. This may also happen when an obsolescent product is reimplemented: cores are reused unchanged to avoid design costs and compatibility problems. If the SoC uses multiple clocks, scan insertion is complex and may not obtain the desired coverage results. Finally, if the SoC architecture employs three state buses to share resources among modules, quite complex constraints have to be defined for the proper generation of test vectors. These factors may make traditional test methodologies, aimed at stand-alone peripherals, ineffective.

Cores inside a SoC can be tested with either hardware-based [24] and software-based [23] methodologies. In the case of peripheral cores, however, hardware techniques may imply high area overhead, impair performance and even require unacceptably long application times. In addition, testing the SoC at-speed may not be always economically sound using scan chains, but is required to preserve the effectiveness of the test.

An external ATE is supposed to be available for test application: its purpose is to load a test program in the microprocessor 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.

In the case of processor cores SBST has already been used successfully for processor test. On the other hand, application of software methodologies to the test of peripherals

has not been deeply explored. The advantages of software techniques in peripheral test are the same as for processor test: cheap application, at-speed testing, no requirement for additional hardware.

A methodology has been developed for the generation of test sets for peripheral cores starting from the RT-level description of the circuit. The methodology follows a software-based approach, exploiting the processor inside a SoC to test the peripherals. Test generation was performed using an evolutionary tool. The feedback for the tool was made up by the RT-level *code coverage metrics* (CCM).

Several attempts have been presented in the literature to use high level metrics as a proxy for fault coverage measurements. The reason is simple: CCMs on a high-level description of a circuit are faster to extract than stuck-at fault coverage, by orders of magnitude. A gate-level description of a circuit contains much more information than the corresponding RT-level. Furthermore, a fault simulation has to be repeated many times, even using parallel simulators. In addition, techniques based on high-level descriptions can be exploited not only by users, but also by soft-core developers.

However, there is not a clear relationship between CCMs and coverage of the stuck-at faults in the general case. This is especially true when large combinational blocks exist in the circuit, as their testability can hardly be forecast resorting only to high-level metrics. Even if they are inaccurate to get a quantitative coverage measure, high-level metrics can be fruitfully used to drive test set generation.

On the other side, such large blocks are not often contained in peripheral cores, whereas they are present in processors. Thus there is a correlation between CCMs and fault coverage. It is not complete, but strong enough for generation of test sets.

Testing a peripheral core requires specification of both the test program and the input/output data for the peripheral. The base unit of test application is therefore the *test block*. This is defined as a basic test unit 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 external test machine.

The methodology has been originally developed as a manual test generation technique [25]. The process is performed by hand and mainly relies on the experience of a test engineer, who must have a deep knowledge of the peripheral high level description. The goal of the test engineer is to produce a set of test blocks in order to maximize the CCMs.

The order in which the CCMs will be maximized must also be selected. In this work every test block follows a quite rigid framework that allows an easy configuration of the peripheral cores in all possible operation modes by only changing a few parameters; and additionally, to set up a carefully chosen stimuli-set based on the specific test block goals.

The process goes as follows: in the first step, an initial test block is generated, based only on functional information about the targeted peripheral, to maximize the statement coverage; then, the generated test block is simulated using a RT-level description, gathering the first code coverage metrics figures. Based on the obtained information, additional test blocks are generated, with the goal of maximizing the rest of the chosen metrics.

Once the first coverage metric is saturated, another one is tackled in order to increase

the testing capabilities of the final test set. This process is repeated until sufficiently high coverage values are obtained for all the chosen metrics. Remarkably, it must be taken into consideration that for different peripherals the metrics chosen as critical may also differ, as well as the number of considered CCMs for sufficient code coverage.

The test engineer produced every test block based on a quite rigid framework that allows her to easily configure the peripheral cores in all possible operation modes by only changing a few parameters; and additionally, to set up a carefully chosen stimuli-set based on the specific test block goals.

The methodology has been subsequently automated and enhanced through the use of the μGP^3 evolutionary tool. The goal was to enhance the obtained results and, especially, to reduce the dependence of the methodololgy from the knowledge of the test engineer.

Test set generation is performed with the goal of maximizing the CCMs. Some code coverage metrics suitable for guiding the development of the test sets for peripheral cores embedded in a SoC are: *statement coverage* (SC), *branch coverage* (BC), *condition coverage* (CC), *expression coverage* (EC), *toggle coverage* (TC). These are listed in the order in which usually it is most effective to maximize them.

Many authors hold that it is not possible to accept a single coverage metric as the most reliable and complete one; thus, a coverage of 100% on any particular metric can hardly guarantee a 100% coverage on the stuck-at faults [26]. 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. It is essential to carefully choose the set of metrics to be maximized to reduce redundant efforts on the test set generation.

To perform the generation process, an automatic incremental methodology has been implemented by which test blocks are iteratively generated, that collectively obtain the desired high-level metrics coverage. Clearly, the configuration code fragment, functional fragment and stimuli set composing a test block must be generated concurrently, in order to maximize the profits obtained by each test block. The generation process ends when either the targeted CCM figures are satisfactory or after a predefined time limit, producing one test block for every performed step.

The evolutionary core acquires information regarding the SoC under evaluation from its constraint library. It describes the syntax of the processor core and provides information about peripheral constraints, such as data length and controlling information, such as external signals and configuration words. The constraints concurrently define two different entities, the functional part and the external data. Each individual, which represents a test block, is then composed of a test program and a set of stimuli; it is evaluated by a high-level simulator. Results obtained from the simulation are fed back in the form of a fitness value to the evolutionary core closing the generation loop.

Since the μGP^3 is able to receive multiple feedback information for a single test block, it can simultaneously maximize all the chosen coverage metrics. The relative importance of the coverage figures is enforced simply by the order in which they are communicated to the tool.

The methodology has been applied on a simple, purposely designed SoC. This includes

a processor core compatible with the Motorola 6809, a serial *universal asynchronous receiver and transmitter* (UART), a parallel *peripheral interface adapter* (PIA), and a RAM memory module. The system is derived from one freely available on an open source site [22].

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.

Measure	**PIA**	**UART**
statements	149	383
branches	134	182
conditions	75	73
expressions	0	54
toggle bits	77	203
gates	1,016	2,247
faults	1,938	4,054

Table 2.11. Implementation statistics for the SoC

Table 2.11 shows details of both peripherals the PIA and the UART, including information at high and low-level. 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.

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 in polling or interrupt mode, and with parallel data communication (transmit and receive) with different control schemes. In the case of the UART, available configurations include polling or interrupt mode, serial data communication (transmit and receive) with different data bit numbers, with or without parity, with 1 or 2 stop bits, and with different communication rate ratios.

To more thoroughly assess the 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. The manual method has completely been implemented by an expert test engineer.

Both test sets have been assessed performing a gate-level fault simulation after generation. No data from the gate level, in contrast, is used to drive the generation.

The evolutionary methodology proceeds as follows. For every peripheral module the description style is analyzed to choose the most critical coverage metrics and their relative importance. A first test block is then generated and evaluated for code coverage. If the attained results are not satisfactory, more test blocks are generated, but the measurements are modified in order not to take into account the previously covered parts. For example, for statement coverage only the newly covered statements are considered and maximized, neglecting the already covered ones.

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 code resembles 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. 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 to send the test results to the ATE.

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. 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.

The evolutionary tool uses 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 order to simultaneously optimize all of them. The order of the metrics was SC, BC, CC, EC, TC.

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.

The main evolutionary parameters have been set differently for the PIA and the UART. For the PIA they were $\mu = 50$ and $\lambda = 70$. As the UART is more complex than the PIA simulations take a longer time, so the parameters were kept lower, at $\mu = 30$ and $\lambda = 40$. Clone extermination was used, as well as a fitness entropy hole of 1.0.

	PIA manual	**PIA EA**	**UART manual**	**UART EA**
Steps	8	5	7	3
SC	100.0%	100.0%	95.5%	100.0%
BC	96.9%	96.9%	92.9%	98.9%
CC	89.3%	90.7%	97.3%	98.6%
EC	NA	NA	72.7%	94.5%
TC	100.0%	100.0%	89.2%	100.0%
FC	89.78%	90.20%	80.96%	91.43%
Test time (ck)	800	5,800	5,000	101,685
Generation time	30h	2h	60h	1.5h

Table 2.12. Comparison between manual and automatic results

In table 2.12 the results for the manual and automated methodology are compared, both in terms of coverage and with respect to the times needed for application of the methodologies.

For the PIA, the manual methodology produced 8 test blocks, occupying about 200 bytes in program code and employing 19 data, whereas the evolutionary method generated 5 programs, occupying about 480 bytes, and 80 input data.

For the UART, the manual technique ended with 7 programs, about 250 bytes long, and 10 input data. The automatic technique generated 3 test blocks, including about 1,380 bytes of code and 231 input data.

The EA is also able to reduce the generation times by about an order of magnitude.

These results, most importantly, experimentally prove that a correlation between high-level coverage metrics and stuck-at fault coverage indeed exists, since an increase in the first group of metrics is followed by an increase in the second.

The main drawback of the automated methodology is that a high setup time is still needed. This happens because, for each peripheral, a complete set of program templates has to be written and subsequently included in the constraints library for the tool. This not only requires time, but also the work of a skilled test engineer, and a deep knowledge of the modes of operation for the target peripheral. One of the most challenging requirements of the manual method, therefore, is not removed.

To overcome the need for deep knowledge of the peripheral core a relaxed evolutionary methodology has been implemented. The basic framework closely matches the above description, but the constraints library for each peripheral no longer contains a separate framework for each main mode of operation. Only two general operation modes are kept, namely polling and interrupt, since the program structure changes between the two. In contrast, no configuration information is included to specify transmission rates, formats, or the like.

This means that the evolutionary algorithm must be able to autonomously generate the code for correct exercising of the peripheral. Likewise, the data section of the library has been reduced, eliminating the neat distinction between data and control signals. A parametric delay was left in the library to allow respecting timing constraints.

The experiments have been repeated, with the same multi-run incremental scheme and preserving the order of the CCMs used. The evolutionary parameters have also been preserved from the previous methodology.

Table 2.13 reports the best results obtained with each of the two methodologies. In this table PIA rig. and UART rig. refer to the rigid evolutionary methodology, whereas PIA rel. and UART rel. refer to the relaxed methodology.

Table 2.14 shows that in both evolutionary experiments the memory occupation and application time of the set increase; however, it is also possible to note that the elaboration time required to devise the complete test set is strongly reduced.

Repetition of the experiments and close analysis of the results allowed discovering that a significant variation exists within them. In fact, differences of as much as 13% in fault coverage were detected on the UART, in contrast with no more than 3% on each of the high-level metrics.

This means that the correlation between the two kinds of metrics, though useful, is

	PIA rig.	PIA rel.	UART rig.	UART rel
Steps	5	3	3	2
SC	100.0%	100.0%	100.0%	100.0%
BC	96.9%	97.7%	98.9%	98.9%
CC	90.7%	90.7%	98.6%	98.6%
EC	NA	NA	94.5%	94.5%
TC	100.0%	100.0%	100.0%	100.0%
FC	90.20%	91.95%	91.43%	90.68%

Table 2.13. Comparison between rigid and relaxed methodologies

	PIA rig.	PIA rel.	UART rig.	UART rel.
Test time (ck)	5,800	9,506	101,685	28,842
Setup time	54h	3h	60h	3h
Generation time	2h	3.2h	1.5h	2.1h
Memory (bytes)	480/80	1,572/93	1,380/26	1,953/72

Table 2.14. Comparison between manual and automatic results

not strong enough to give the user the confidence that the generated test sets are of good enough quality.

The methodology has again been modified to address this problem, searching for a metric that more accurately reflects the actual fault coverage.

Whereas the ideal metric to exercise all the system functions would be the path coverage (PC), it lacks an explicit correlation with data. Moreover, its use is not practical: enumeration of all the possible paths has exponential complexity and it is not feasible to cover them within a reasonable amount of time.

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 finite state machine (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.

The approach is then based on modeling the entire system as a FSM which is dynamically constructed during the test generation process. Thus the FSM extraction is mostly automated, and requires minimum human effort: the approach only requires the designer to identify the state registers in the RT-level code and does not require a deep knowledge

of the peripheral and its hierarchy between modules.

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 no approximate metric can be taken as the only important one, maximizing more than one metric usually leads to better quality tests.

Thereby, the developed methodology exploits concurrently all the available CCMs to thoroughly exercise the peripheral functionalities and the FSM transition coverage that enforce a maximum interaction between peripheral modules.

The generation of test blocks follows again a relaxed approach, in which the evolutionary tool is left the task of discovering the optimum operation sequences for the peripheral.

The previously used test case has been augmented with the introduction of a *video display unit* (VDU), able to provide the system with a text-based interface. It manages a display area 80 characters across by 25 down, contained in a text buffer memory of 2kiB and a character attribute memory of 2kiB.

Measure	**PIA**	**VDU**	**UART**
statements	149	153	383
branches	134	66	182
conditions	75	24	73
expressions	0	9	54
toggle bits	77	199	203
gates	1,016	1,321	2,247
faults	1,938	2,234	4,054

Table 2.15. Implementation statistics for the SoC

Table 2.15 reports the implementation statistics for the three considered peripheral cores.

The FSM is dynamically constructed starting from the RT-level simulation. RT-level code must be instrumented to make the value of all state registers observable. The evaluator of the evolutionary tool collects this and filters it in order to identify the global state of the peripheral; every global state in the peripheral represents a possible configuration of values of all the state registers.

Thus, whenever, in the simulation, a state register in any module changes its value, the global state of the peripheral is affected; every newly discovered transition between the global FSM state is stored in a hash table, for efficiency. At the end of the simulation the number of different keys in the hash table represents the number of transitions. This is fed back to the evolutionary tool together with the traditional CCM figures, also available from the RT-level simulation.

The number of possible transition could be very high, but this does not impair the methodology. The FSM extraction is dynamic and unsupervised, and only effectively

explored transition are taken into account.

Table 2.16 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 in percentage. It is important to note that the value of traditional CCMs are expressed as absolute values, instead of percentages.

	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%

Table 2.16. Results for the considered peripherals

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. But, since the FSM extraction is dynamic, this did not lead to any additional problems.

To experimentally demonstrate that the use of the FSM transition coverage allows strengthening the correlation between high an low level metrics 100 experiments on the UART have been performed, using both the relaxed methodology without using the FSM coverage and the generation process augmented with it.

	Max	**Min**	**Average**	**Std. dev.**
Statement	383	381	381.8	0.36
Branch	180	178	178.7	0.39
Condition	72	70	70.7	0.30
Expression	51	50	50.7	0.32
Toggle	203	201	201.3	0.40
FC	90.7%	77.0%	84.8%	6.37%

Table 2.17. Results without the FSM coverage

Table 2.17 reports the results of the experiments performed. The table illustrates the average and the standard deviation of the different coverage metrics (Statement, Branch, Condition, Expression, Toggle) and of the stuck-at fault coverage (FC). It is worth nothing that the CCMs are very near to the absolute maximum with a little standard deviation while the fault coverage has a greater standard deviation. Even if the values of the CCMs are about the same among all the experiments, the standard deviation in the fault coverage of each test set is relatively high. A large standard deviation implies that the distribution of the solutions found is sparse so there is a considerable difference in fault coverage

between the test set found: the methodology, although it obtains good results, is not as robust as desirable, and the obtained solution may not exhibit the expected quality.

	Max	**Min**	**Average**	**Std. dev.**
FSM transition	142	138	141.0	1.49
Statement	383	382	382.2	0.28
Branch	180	178	179.3	0.33
Condition	72	71	71.8	0.22
Expression	51	50	50.8	0.24
Toggle	203	201	202.2	0.36
FC	91.3%	89.6%	90.9%	1.10%

Table 2.18. Results using the FSM coverage

Table 2.18 shows the results of the experiments performed using also the transition coverage on the peripheral FSM as a high-level metric; the table reports the average and the standard deviation of the different coverage metrics (Statement, Branch, Condition, Expression, Toggle) and the stuck-at fault coverage (FC) as in table 2.17 in order to make a fair comparison.

It could be noted that the CCMs are slightly but consistently better than those reported in table 2.17 and the average fault coverage is increased by more than 6%. Much more importantly, the standard deviation of the fault coverage is dramatically reduced. This means that the robustness of the methodology has been increased, and solutions of consistent quality can be obtained.

The methodology is able to find test blocks that, on average, reach nearly the maximum possible value of CCMs at RT-level and, above all, have a similar fault coverage at the gate level. This experimentally demonstrates that the correlation between high-level metrics and low-level one has been improved.

Table 2.19 reports a comparison between the two methodologies in the case of the UART, in terms of obtained fault coverage (FC), average generation time (Tgen), average application time (Tapp) in clock cycles, and average size of the test sets, reported as program bytes and data bytes.

	FC	**Tgen**	**Tapp**	**Size**
Without FSM	90.7%	5.1	28,842	1,953/72
Using FSM	91.3%	2.2	32,762	2,345/87

Table 2.19. Methodology comparison for the UART

It is worth noting that the generation time also benefits from the use of the additional metric, even if it was not one of the pursued goals.

2.4 Compaction of existing test programs

SBST methodologies, especially manual ones, often have as a result one or more test programs organized in the form of one or more nested loops. Well-known methodologies [4] are exemplar in this. The loop structure is natural for these programs, since they are targeted towards specific processor blocks. One or more *feeder* instructions are selected among those provided by the instruction set architecture (ISA) of the processor that use the target block. The feeder instruction is then put inside one or more nested loops to provide the test patterns to the block. The control part of the loops is in charge of generating the patterns.

Such methodologies are very general and highly portable, since the original test programs [5] can be adapted to a different processor by translation of the assembly instructions. This generality, however, has a drawback: loop-based test programs are often very compact in memory but redundant in the number of executed instructions. For very small microprocessors it is possible to test the arithmetic blocks exhaustively, due to the low parallelism, whereas for larger machines this is totally infeasible.

To contain the complexity and length of these test programs usually the available parameters are the loop extremes and the stride. More test patterns generally mean better fault coverage, but also longer test application times. Even when the program does not simply implement an exhaustive test, the test program strikes a balance between the number of test patterns, their effectiveness, and the algorithmic complexity required to generate them. This balance may not be optimal, or the only one possible.

It can be experimentally proven that, in such programs, not all the input patterns provided by the feeder instruction are actually necessary to obtain the final fault coverage. Many of the values provided to the block under test turn out to be just convenient intermediate steps between really useful test patterns.

It is possible to prove the above assertion using an analytic methodology that decomposes an arbitrary test program into a large number of very small, independent code fragments. These fragments are called *spores* to emphasize their small size and the fact that each one is targeted towards a single specific operation. The purpose of a spore is that of reproducing, as exactly as possible, the state transitions that occur during execution of a single instruction in a test program, and then make the result visible to the outside world. The instruction whose execution is to be reproduced is called *objective* instruction.

The structure of the spore follows directly from its function. An initial instruction sequence puts the processor in the state existing before the objective instruction, then the objective instruction is executed, and finally further code propagates the results to the external world, using either the processor ports or the external memory. The initial state includes the values in the source registers of the objective instruction, so that it is executed with the correct operands.

The spore concept was initially developed for non-concurrent on-line test applications, but it is a powerful analytical tool to get insight in the operation of a test set. In particular it allows measuring the contribution of every single instruction to the final fault coverage, at least for data processing instructions targeting the data path blocks.

Spores can be useful for diagnosis, as their low fault coverage allows partitioning the

fault universe in small sets. They could even be useful for silicon debug, since they show the results of single instructions executed in specific conditions.

The generation of a set of spores from a program, also called *sporification*, is performed through the use of a tool, named *frantumator*, specifically developed for the target processor [11]. It is essentially an instruction set simulator (ISS) augmented with the capability of generating a small test program for every instruction executed.

The sporification process performs a logical decomposition of the original program, generating a self-contained test program for every instruction execution. This allows to evaluate the effect of every single instruction instance.

The only real drawback of sporification is that it generates a very large number of programs. It is easy to obtain tens of thousands of spores from the decomposition of a test loop.

Redundance in the loop-based test programs was shown in the following way. First, the program has been decomposed in spores. Then those corresponding to the feeder instruction in the loop have all been fault simulated, to obtain the necessary coverage data. To avoid measuring the fault coverage due to the initialization or propagation parts of the spore, a test program has been chosen that targets a block not used by any instruction in the spore except the objective instruction, and only the faults in the target block have been taken into account. In this way it is ensured that only faults covered exclusively by the feeder instruction are accounted for.

The second step has been a greedy set covering to obtain a minimal set of spores that obtain the full fault coverage of the original program on the target block. The reason for the use of a greedy algorithm instead of an exact one is that set covering is an NP problem, so the exact solution could not be found in reasonable times. This is, however, not an issue because the solution found perfectly illustrates the point: less than 2% of the original feeder instructons were actually needed to achieve the full fault coverage.

Compaction of test patterns for digital crcuits is a widely investigated field. Indeed, test application time is one of the most critical metrics for a manufacturer, since it directly reflects on product cost. For on-line applications, conversely, test program size may be as important as application time, especially for embedded applications, where memory is at a premium.

For combinational test, there is a direct relationship between test size and application time, so the two goals go hand in hand, and efforts are spent only for elimination of the redundant patterns. For scan-based methodologies the picture is more complex, since a large part of the appication time is consumed by the operation of the scan chain itself. A careful choice of the pattern sequence, and optimal use of the scan chain, can improve test application time as much as discarding redundant patterns. It may even be more time-effective to keep more patterns, that can be applied with a limited usage of the load/unload operations, than to use a minimal pattern set [] [].

Test programs cannot be compacted in the same way, for various reasons. The first is that a test program generally contains flow control instructions. These include loops, conditional execution of data processing instructions, jumps to subroutines, and so on. Test length is no more related to test application time, not even in a rough manner.

Test program compaction has been investigated by the research community. A methodology [9] has been presented to manually generate test programs suitable for microprocessor validation. The methodology is based on the the definition of test specification expressions (TSE), used to describe the properties desired for test programs. The procedure leads to the generation of a large number of test programs. A subsequent procedure tries to merge programs with similar instructions. The main drawback of the compaction methodology is that it is heavily dependent on the TSE, so it is not easily applicable to programs generated with differnt methodologies.

Differently from the case of combinational or scan-based test, test programs cannot usually generate an arbitrary input sequence. Inputs for one instance of the feeder instruction are derived functionally from the previous ones or from loop counters. Once the complexity of this function becomes greater than a certain threshold, it becomes cheaper and more convenient to store the input values as data inside the test program itself, losing the advantages in small size that SBST may provide. Similarly, a test program obtaining full fault coverage might be obtained, after the greedy set covering, simply joining together the corresponding spores, but it would be even longer.

To make a loop-based test program execute in less time while preserving its fault coverage it is necessary to change the function that generates the input patterns. Different values should be generated, so that the full fault coverage is obtained with less patterns. Small program size and short application time are in general conflicting goals.

The methodology described below is based on three steps: decomposition of an existing loop-based test program into a spore set; fault simulation of the spores corresponding to the feeder instruction; aggregation of a suitable subset of the spores to form again a loop-based test program. The goal of this methodology is to preserve the compactness of the original program while reducing the loop count. It is important to note that, due to reasons exposed above, the subset selected may well not be minimal.

To be useful, the resulting loop must satisfy three minimum requirements: it must be smaller than the sum of the spores composing the minimal covering set; it must take less time than the original loop; it must generate a sequence of values able to fully cover the faults detected by the original program.

Since the functional dependence between an input pattern and the next cannot be arbitrarily complex the spore subset is not directly selected. Instead, a couple of functions is built, able to generate a subset of the input patterns with the desired properties. In this way it is possible to simultaneously take into account the fault coverage obtained and the complexity of the functions.

The two functions to build are $f_a(a) : \mathbb{N} \to \mathbb{N}$ and $f_b(b) : \mathbb{N} \to \mathbb{N}$, depending on two variables. These functions are to be used inside two nested loops in a way that mimics the structure of the original program. $f_a(a)$ and $f_b(b)$ must be able to iteratively generate a sequence of value pairs for the feeder instruction that allows reaching the full fault coverage. Each pair P_i corresponds to a set C_i of covered faults. The union of all sets C_i gives the total fault coverage of the program generated in this way.

Two points are worth noting now: the first is that there is no particular requirement on the order in which the value pairs should be produced within the loops; the second is that, following this methodology, only value pairs already computed by the original

program are associated with a nonzero fault coverage, while others are deemed useless, even if they are not. This last consideration has further implications. One is that the greatest potential for optimization is given by the most redundant programs, since many pairs are associated with a non empty coverage set, giving more opportunity for efficient coverage. Another is that, given a test program, the fault coverage on the target module may only increase or remain unchanged at the end of the process, but not decrease.

The search space for the problem is large. The original program feeds n input patterns to the target module. These are generated in a nested loop, so $n = n_a n_b$, where n_a is the number of distinct values for the first operand and n_b is the number of values for the second. The compacted program has to produce $m = m_a m_b$ value pairs, $m < n$. Since there is no restriction on the ordering of these patterns, they may be generated in $m_a! m_b$ different ways. Each sequence can be produced by one or a few minimal-cost functions. The total number of function that may solve the problem is therefore huge.

To limit the complexity of the problem only functions that use integer and logic operations have been used. In particular a small, functionally complete set of operations has been selected, including addition, subtraction, bitwise AND, bitwise OR, XOR, shift, binary negation, least significant byte, most significant byte. This choice has several advantages. It avoids all the approximation problems associated with the use of floating point arithmetc, both in the intermediate computations and in deciding whether a given value has actually been produced. It also avoids the computational cost of floating point arithmetic, usually higher than that of integer operations. Most importantly, the chosen operations are easily mapped to assembly language instructions. The generated functions can closely match the desired result, fragments of code computing the sequence of input patterns. This close match makes computation of a cost metric easier.

The methodology was applied to compact a post-production test program written for an i8051-compatible microcontroller. The microcontroller is available as a synthesizable VHDL description, freely downloadable from the *opencores* site [22]. A fuller description of the microcontroller is contained in section 3.1.

A small infrastructur IP (IIP) is supposed to be available and connected to the processor ports. Its purpose is to collect an incremental signature of the test program, as it repeatedly outputs the intermediate results of the computation. The signature is collected through the use of a 24-bit linear feedback shift register (LFSR).

The program has been written according to a methodology similar to that exposed in [4], and targets the multiplier unit. This multiplies two 8-bit inputs and produces a 16-bit output. In this program the feeder instruction is the MUL operation, that multiplies the content of the A and B registers, storing its result in the same registers. The inner loop is repeated 2,048 times, providing the same number of input patterns to the multiplier. The attained fault coverage is 82.9% on the multiplier module.

All fault simulations have been performed using a fast, in-house developed fault simulator. It has been used to contain times, since the number of fault simulations to perform is very large, and the corresponding times very long.

The sporification process has been carried out using another home-made tool, since no available open-source or commercial tool performed the required functions. The generated spores set the input registers and the program status word to the values they had during

program execution, execute the objective instruction and then write its output values and the new program status word on the processor's output ports.

The spores containing the multiplication instruction have been selected and fault simulated. This stage provides all the sets of covered faults to be used in the following steps.

For the final step the μGP version 2 has been used. The revision used features self-adaptation, local mutation, individual aging and selectable elite size, but not the scan mutation operator.

The reasons for the use of an evolutionary approach are several. First, there is not a simple relationship between a possible solutin to the problem and its effectiveness. Solutions in this case are the representations of the f_a and f_b functions. Although the single operators composing the functions are simple there is no predefined limit to the complexity of the resulting program behavior [32]. The sequence of values produced can be seen as the emerging behavior of the operators that compose the function. An evolutionary approach can evolve a function with arbitrarily complex behavior, meaning that the full fault set can be covered, no matter what spores are needed to achieve the result.

The evolutionary tool has been used to evolve the dataflow graph of the functions. In the graph every vertex can contain a constant integer value in the range $[0 \dots 255]$, a unary operator or a binary operator. Two additional constant values, in the same range, are associated with each function. These are the initial values for the a and b variables, and the number of iterations for each of the nested loops. In sort terms, the evolutionary tool has been left the widest possible freedom in evolving the solutions.

The fitness is computed by a script that parses the graph, executes the two nested loops the prescribed number of times while computing a cumulative fault coverage. No new fault simulation is needed at this stage because the data computed for each spore is used. If input patterns outside the set generated by the original program are produced their contribution is not considered, as if they covered no fault. The pseudocode of the coverage evaluation is as follows.

```
a=initialValueA;
do{
  b=initialValueB;
  do{
    evalFeederInstruction(a,b);
    onePass(b);
  } while (!bCompleted());
  onePass(a);
} while (!aCompleted());
return (coveredSet);
```

The fitness is composed of two terms. The first one is the fault coverage attained by the current solution, and its upper bound is the full fault coverage of the original program. The second term is a cost metric, computed as the sum of the computational cost for every evaluation of f_a and f_b. The first term has the priority over the second, expressing the fact that preservatin of the fault coverage is a hard bound for the problem.

Two series of experiments have been performed to search a suitable solution. A $(\mu+\lambda)$ strategy has been used, since a complex structure of the solution was expected. The population parameters were set at $\mu = 500$ individuals and $\lambda = 500$ evolutionary operators per generation. In the first series the evolution has been carried out for 200 generations. In the second series of experiments two normal evolutionary steps have been executed, separated by an intermediate relaxation step. In the relaxation step the two fitness terms are exchanged. The number of generations in the second series is larger, comprising 150 generations for the normal steps and 10 for the relaxation step.

The population parameters have been chosen based mainly on previous experience in the use of the evolutionary tool, with the goal of maintaining sufficient diversity in the population to avoid premature convergence.

The experiments with the relaxation step were aimed at the same goal. A normal evolutionary process might have converged to a local optimum, and the purpose of the relaxation step is that of escaping that optimum, without losing too much in terms of quality of the generated solutions.

For comparison an individual exactly corresponding to the original program has been written manually. It is named *original individual*, although it does not necessarily correspond to any individual generated during evolution. The original program does not perform an exhaustive test, but executes less operations for speed reasons.

The evolutionary process is able to share vertices and edges of the dataflow graph between the two functions. In this way, if a link between the two functions is useful, it can be transmitted from an individual to its offspring, even if the shared components are mutated.

Individual	Size	Iterations	FC	Cost
Original	3	2,048	100.0%	16,408
Sample	5	6,630	100.0%	53,235
Best	3	600	100.0%	4,216

Table 2.20. Comparison between individuals

In table 2.20 the final statistics for the experiments have been collected. The table reports the indication of the individual, its size in assembly instructions for the computation of the functions, the number of iterations of the feeder instruction, the fault coverage relative to the original program, and the cost metric for the programs.

The results show a 74.3% reduction in computational cost is achievable with respect to the original program without any penalty in fault coverage. The best solution found, furthermore, is as long as the original program, so no additional memory is necessary.

Some of the basic assumptions turned out not to be true. Indeed, the resulting functions were expected to be significantly more complex than the original ones, which are very simple. On the contrary, no increase in complexity was found in the final solution, even if some intermediate individuals were actually very complex. Also, the second series of experiments did not obtain better results than the first.

Chapter 3

Diagnosis Activity

Diagnosis, like test, has been theoretically investigated in the past, and is a standard industrial practice. Like test, diagnosis is an integral part of the design and manufacturing process. The roles of the two activities, however, are different. This difference is reflected in the amount of dedicated hardware support: extensive test structures are commonly added to a processor whereas little hardware is inserted specifically for diagnosis.

Even more than test, diagnosis is a costly activity, both for generation and for application. As for test, software-based methods can decrease the application costs, but the problem of diagnosis set generation is not solved.

The work in diagnosis has been focused on the definition of general methodologies for the generation of effective diagnosis sets. Some of the main goals for these methodologies have been economic viability and scalability, so they can be applied to a large range of processor cores.

3.1 Microprocessor Diagnosis

A complete software-based methodology for the diagnosis of microprocessor cores has been developed. It starts from existing test programs, and can exploit a small, already existing, infrastructure IP (IIP) to perform the diagnosis.

The basic idea behind the methodology is that a processor, once developed, is accompanied by a set of test programs able to reach a high fault coverage. Since diagnosis can only be performed using structural information, it is an activity performed almost exclusively by the manufacturer of the processor core. Usually the manufacturer generates, during development of the circuit, a series of programs for validation, verification and then test of the device. Simply discarding them because they are probably not good diagnostic programs seems wasteful.

Indeed, the generation of a test set with high fault coverage implies a high effort, either manual or computational, to detect the subset of hard to test faults. Neglecting the existence of these programs means that the same efforts would need to be repeated to cover the very same faults with the diagnosis set.

Software based diagnosis has its own special set of problems when compared to hardware methodologies. First, there is usually a common part of logic, including at least the fetch logic and part of the decoder, excited each time one instruction is executed, since test programs rely on instructions rather than on test patterns. Additionally, a test program suited to excite a specific module of the processor could also cover a wide number of faults not belonging to the pinpointed part, and therefore offer a very reduced classification ability. Finally, many of the internal processor circuit elements cannot be accessed directly using a specific instruction, thus resulting hardly diagnosable.

These problems are exhacerbated when using an effective test program for diagnosis. Test programs cover large sets of faults, but are not usually written to distinguish them. Indeed, the constraints in terms of application time and memory size for test programs forbid the pursuing of this goal. A better diagnostic capability could be obtained by writing a large set of programs each of which covers a few faults [13].

Usually the high fault coverage is obtained by the stimulation of the functional modules of the processor with a great amount of input data, generated in a looping section of the test program. Many faults, especially in arithmetic units, need specific bit patterns on the execution unit inputs to be covered. If a program can be written that delivers only those data needed to detect a specific fault, and not others, that program would exhibit a very low fault covering ability, but a fairly good diagnostic capacity.

The real problem becomes then that of enhancing the diagnostic capability of a relatively small set of test programs, by transforming them in a much larger set. This is obtained by a decomposition of the test program using an analytic methodology. The purpose of this methodology, called *sporing*, is the decomposition of an arbitrary test program into a large number of very small, independent code fragments [11]. These fragments are called *spores* to emphasize their small size and the fact that each one is targeted towards a single specific operation. Each spore reproduces, as exactly as possible, the state transitions that occur during execution of a single instruction in a test program, and then make the result visible to the outside world. The instruction whose execution is to be reproduced is called *objective* instruction.

The sporing tool is essentially an instruction set simulator (ISS), that traces the processor state for every instruction executed. It is augmented with the capability of producing a program that, for every instruction in the test program, is able to infer each instruction data flow graph. From this the sporing tool generates an independent fragment of code able to replicate the processor behavior during that particular execution. Sporing is applied to the *initial test set*, or *post-production test set*, generating a *spore diagnosis set*.

It is of particular importance that the spores are independent from each other. Each one has to be a self-contained program, and should ideally need no particular support. Secondly, and more importantly, it must be possible to execute the spores in any order to perform diagnosis, so the execution of every one must be independent from the execution of any other code.

The structure of a spore is simple, performing just three operations. The first is controlling the processor. This means setting the processor state, including the operands of the objective instruction and the flags of the program status word (PSW) to the value they had before the execution of the instruction in the test program. The second is the

execution of the objective instruction. The third is the propagation of the results of this execution to an observable location, such as the memory or the processor ports.

In the case of pipelined processors, a spore generated for an objective instruction should include an initialization sequence able to reproduce the exact state of the pipeline stages. This is possible by including in the spore code all those instruction preceding the target one and still influencing the pipeline stage state during its execution.

One noteworthy point is that the sporing process does not depend on the methodology used to write the test program, since no assumption is made about the code. Another important consideration is that, since the spores are not generated from scratch, but obtained from an existing test program, the global fault covering ability is preserved.

Two types of spore may be generated by the sporing tool. There are *executive spores*, originated from data processing instructions, performing arithmetic or logic operations, or memory transfer instructions. There are additionally *branch spores*, originated from instructions performing jumps in the initial test program.

Spores have some minimal test capability and own the nice property of being small, in terms of both code size and test duration. The spores set reproduces totally the execution of the initial programs, thus preserving completely its fault coverage. The number of spores obtained by the fragmentation of each test program increases linearly with the number of instructions executed by the original program.

Once the spores are generated it is necessary to perform a fault simulation for each unique spore. In this way the set of faults covered by every spore is found. The result of this step is a *coverage matrix* storing for each spore the information about covered faults.

The only serious drawback of the sporing process is that it produces an inordinate number of programs, since a spore is produced for every execution of every instruction in the test set. It is not exceptional to obtain a spore diagnosis set comprising tens of thousands of different spores, derived from the iteration of loops inside the test set. Application of all these programs to a processor for diagnosis purposes would take an unacceptably long time. An effort to reduce this diagnosis set is therefore appropriate.

This goal is obtained through *sifting*. It is a heuristic filtering methodology, whose purpose is to reduce the number of programs in the diagnosis set by eliminating redundant code. Sifting must also guarantee that no fault coverage is lost in the process, and that the diagnostic capacity of the spore set is preserved.

First of all, the fault covering ability of each spore is considered in the context of the entire diagnosis set. The important thing for a spore is not covering a large number of faults, but covering faults which are not detected by other spores: all the spores able to do this have to be retained in the final diagnosis set.

Every fault is detected by a certain number of spores, depending on whether it is easy or hard to detect. This leads to the concept of *density* of the fault, that is, the number of spores able to detect it. The density of a fault F is denoted by d_F.

The diagnostic capability of a spore is then evaluated with respect to the whole set, using the density concept. Indicating with NF_s the number of faults covered by spore s, every spore is assigned a preliminary fitness value

$$f_s = \left(\sum_F \frac{1}{d_F} \right) \cdot \frac{1}{NF_s}$$

The possible vaules of f_s range from 0 to 1. Higher values correspond to better diagnostic capability of the spore. A spore can have fitness 1 if and only if it covers only faults that are not covered by any other spore. In general this fitness measure rewards spores that cover faults that only a few other spores cover. Every such spore must undergo a probability greater than average to be included in the final set.

For sifting the spores are sorted by fitness value in decreasing order. After this is done, a sequential selection process begins: starting from the top of the list, spores are kept until their cumulative fault coverage equals that of the post-production test set, and the others are discarded as redundant. These selected spores form the *basic* diagnosis set. The pseudocode for the sifting phase is as follows

```
foreach (s in SporeSet)
  evaluateSporeFitness(s);
sortSporeSet();
T=faultCoverage(SporeSet);
foreach (s in SporeSet)
  B:=B+s;
  exit if (faultCoverage(B)=T)
```

After the sifting phase, the diagnostic ability of the set is evaluated. One of the major costs for such evaluation activity consists in the computational effort required by fault simulation experiments. For processors, the fault simulation setup is often a severe bottleneck in test programs generation. In order to reduce the impact of this computational cost, the illustrated procedure for diagnosis set generation considers two kinds of fault simulation. The first is *pass/fail fault simulations*, stopped as soon as a faulty behavior is observed and providing pass/fail information. The second is *complete fault simulations*, which require the execution of the entire test program and returns the faulty circuit syndromes. These two differ in required computation time and in the results obtained.

Pass/fail simulation only provide a single bit of information for every fault at the end of program execution, allowing a *coarse* classification. In contrast, complete fault simulation provides several bits of information for every fault and test program, allowing to better distinguish faults, leading to a *fine* classification.

Such a differentiation allows implementing a tree-based classification methodology that may requests for many “fast” pass/fail fault simulations, but leads to the need of only few “slow” fine fault simulations. The subdivision of the process in two steps reduces the total computation time needed.

The coarse classification is based on the construction of a compact diagnostic tree obtained by processing only the pass/fail information related to each test program included in the initial test set; this information is extracted from the coverage matrix.

The fine classification is then performed for all the equivalence classes isolated by the coarse classification and still composed of more than one fault. This second classification is generally done by using the faulty circuit responses on primary outputs, or the *syndromes*, to build in parallel an n-ary tree for each EC to be further divided.

To reduce the impact of this second fault simulation process aimed at retrieving the faulty syndromes, an incremental approach has been used. First, the n-ary tree structure is updated at the end of the fault simulation of each test program in the initial diagnosis set and faults included in ECs of size 1 permanently dropped from the fault list. Second, faults not yet classified and not covered by the next test program to be fault simulated are temporarily dropped from the fault list as their syndrome is not useful for classification.

This process allows the generation of a compact fault dictionary composed of a pass/fail sequence leading to a coarse EC and the set of discriminating syndromes for each fine EC.

At this phase of the workflow, there is still a set of unsatisfactorily large ECs. Thus, an effort is appropriate to partition these large classes.

To improve the diagnostic ability of the test set an improvement step has been performed using the μGP evolutionary tool. It is an evolutionary approach to generic optimization problems with a focus on the generation of test programs for microprocessors. It is based on an evolutionary core, an instruction library that allows targeting a specific microprocessor and an external evaluator to provide the core with the necessary feedback.

The evolutionary approach receives the information about the large ECs and generates new assembly programs able to split them. The fitness function provided as feedback to the evolutionary core is the size of the largest EC found using the fine classification. During this phase, the ability of a generated program in dividing the large EC is directly obtained by analyzing the faulty syndrome. The fine classification tree is directly updated after each program generation. The time taken by the process can be traded with the quality of the obtained results.

The methodology has been applied to a small microcontroller, compatible with the Intel i8051, supposed to be embedded into a SoC. The target fault model is the single line stuck-at.

One of the major problems in testing a SoC is the reduced accessibility of each single embedded core: to improve the observability of the i8051 to be diagnosed, an already developed solution to support the software based self test procedure of processor cores has been employed. The i8051 core was complemented with a 24 bit multiple input signature register (MISR) connected to its parallel output ports [10]. The faulty signature, stored by the MISR can be read at the end of each test program execution.

The synthesized microcontroller, obtained using a generic home-developed library, contains 37,864 equivalent gates, and the collapsed fault list counts 13,078 faults. These faults do not include those inside the RAM module, since specific procedures are customarily used for memory test and diagnosis.

The reference metric used is the diagnostic power for limit n, $dp(n)$. Using this metric $dp(1)$ is the percentage of faults that are uniquely classified; $dp(10)$ is the percentage of faults that can be considered correctly classified, because the exact analysis of equivalence between faults cannot be performed for medium or large sequential circuits.

The sporing process has been performed starting from a manually devised post-production

test. It is composed of 8 test programs written by a skilled test engineer, reaching a fault coverage of about 92% on the collapsed list. 7 out of 8 programs aim at covering the ALU faults and are generated by following the deterministic approach described in [4]; the remaining program has been written resorting to the same technique, but it aims at the coverage of those faults in the decoding and control units still not covered.

The sporing process generates about 60k test programs. Each spore includes an initial and a final part, required to obtain the signature.

The fault simulation process required to proceed to the sifting phase has been done exploiting an in-house developed tool and required about 75 hours on 3 workstations based on SUN Blade processors.

	Post production test set	**Basic set**	**Final set**
Programs	8	7,231	7,266
Test set size	4kiB	165kiB	177kiB
Application time (cl. cycles)	1.00M	1.93M	2.02M
$dp(1)$	11.56%	35.70%	61.93%
$dp(10)$	32.90%	58.02%	84.30%

Table 3.1. Results of the methodology

The sifting phase, including the fault simulation, required about 100 hours on the same hardware. And the test set obtained processing coverage matrix is composed of 7,231 test programs (about 12% of the original set).

Table 3.1 shows the main characteristics of the three test sets: post-production; basic (sifted); final (enhanced). The rows reports: the number of test programs in the test set; the test set size; the maximum length of a test program in clock cycles; the result of the diagnostic assessment $dp(1)$ and $dp(10)$. Table 3.2, on the other hand, details the size of the equivalence classes for the three test sets. The rows refer to the size of the equivalence classes, and the columns report the number of faults belonging to classes of that size for a given diagnosis set. Only faults covered by the diagnosis sets are reported in this table.

It is interesting to note that the initial test set demonstrates a $dp(10)$ equal to 58% of the processor faults. The biggest class, composed of 3,755 faults, is the one including those faults detected by all the test programs of the test set. Apart from this class, the greatest class has size 84.

The diagnostic enhancement of the test set has been performed using the μGPv2 evolutionary tool. This activity eventually led to the generation of 35 new test programs. Each generation process is started by evolving an initial population of 20 test programs. The biggest class obtained by this enhancement process has size 1,092.

If the diagnostic test set is considered as a monolithic block to be entirely uploaded and executed on the ATE, it is composed of all 7,266 programs and requires 2,020,656 clock cycles.

However, if an intelligent/interactive ATE enabling the diagnostic process to be stopped as soon as the fault (or the faults) responsible for the wrong behavior has been individuated, the diagnostic process can be reduced to the execution of an average of 3,762

programs, corresponding to 900,858 clock cycles.

EC size	Initial test set	Basic set	Final set
1	1,334	4,120	7,148
2	802	872	1,106
3	543	522	546
4	360	264	336
5	255	150	155
6	138	150	180
7	112	154	91
8	80	224	72
9	63	81	36
10	110	160	60
11 - 100	2,335	1,090	720
>100	5,410	3,755	1,092

Table 3.2. Equivalence class summary

One of the drawbacks of this methodology is its computational intensity. Generation times total nearly 500 hours of CPU time for all the phases. Another problem is that, despite the sifting phase, the final diagnosis set is large, comprising thousands of programs.

An improved technique has then been developed to mitigate these problems and allow a better scaling of the methodology.

The new generation methodology is based on the existing workflow, but the individual steps have been redesigned to improve results. The starting point, an already existing test program, is the same: the main justification for its use is not just technical but also economic and is independent from the details of the methodology.

The sporing process also remains identical, since its purpose is to decompose an existing program into small fragments corresponding, as closely as possible, to single instruction executions. As the base concept of a spore does not change, neither does the process for their generation. Subsequent steps, however, have been refined.

Two manufacturing test sets, T_M and T_A, were devised to evaluate the robustness of the approach. The first one was manually generated by an expert engineer, whereas the second one was obtained resorting to an automatic tool. The choice of two test sets is intended to evaluate the robustness of the methodoloogy when applied to different test environments.

The manual post-production test set is the same described above. T_A was devised resorting to a completely different method. The approach is based on the μGPv2 tool. In this case, the evolutionary algorithm was exploited in a step-by-step approach: the whole process consists of iterative runs of the evolutionary algorithm, dropping the faults covered at each step. This process generated a set of test programs for manufacturing testing composed of 3 assembly programs. The fault coverage capacity of the set reaches about 90%.

In terms of code organization, T_M and T_A are quite different; T_M is loop-intensive, thus executes many times compact instruction sequences with different operands; on the contrary, T_A sequentially executes a set of selected instructions with suitably chosen operands values and avoids loops.

Such differences in terms of code organization reflect on test application costs. Results gathered by the fault simulation of T_M and T_A underline such an influence, as shown in Table 3.3. In the table test application times are reported in clock cycles. While both test sets attain similar coverage figures, the execution time of the programs contained in the T_M is about 300 times greater than the time of the programs belonging to the T_A; on the contrary, the memory occupation of the set of test programs automatically generated is larger than the space required to save the programs written by the test engineer. The diagnostic ability (shown for the sake of result completeness but not strictly needed in the proposed generation flow) guaranteed by the two test set is low, as could be expected.

Initial test set	**Programs**	**Application time**	**Size**	**FC**	**dr**
T_M	8	1M	3.6kiB	92.52%	92.16%
T_A	3	47.5k	4.3kiB	90.06%	89.17%

Table 3.3. Characteristics of the manual and automatic test sets

At the end of the sporing process, two spore diagnosis sets, sDT_A and sDT_M, are generated from T_M and T_A, respectively.

The new sifting process is as follows: the fitness of every spore is evaluated; then the spore with the highest fitness is considered for inclusion in the sifted set. If the total number of equivalence classes increases then the spore is included, otherwise it is discarded, and the one with the next highest fitness is analyzed.

Every time a spore is included in the sifted set the fitness evaluation has to be repeated, since some faults may be uniquely diagnosed. Taking into account these faults is useless, so they are eliminated from the fault list when computing densities.

```
newClasses:=0;
B:=0;
do
  equivalenceClasses:=newClasses;
  foreach (s in SporeSet)
    evaluateSporeFitness(s);
  sortSporeSet();
  foreach (s in SporeSet)
    newClasses=evalClasses(B+s);
    if (newClasses>equivalenceClasses)
      B:=B+s;
      break;
while (newClasses>equivalenceClasses);
```

Having produced a pass/fail information, every spore has the ability to partition the entire fault set into two disjoint sets. When multiple spores are considered, all these partitions overlap, possibly producing further fragmentation of the fault set.

An increase in equivalence class number directly corresponds to a greater fragmentation. Given an existing partition of the fault set the additional information given by a spore cannot move a fault from a subset, not even that of uncovered faults, to an already existing subset.

Exactly like glass splinters belonging to a slab, fault sets can only be broken up, not merged. Unlike glass splinters, equivalence classes can only be divided up to individual faults, and, more important, they are much more difficult to aim at for splitting.

The advantage of dynamically recomputing fitness is that several spores with similar (static) fitness may help isolate the same subset of faults. Once the first one is included in the sifted set the others become nearly useless, but a static fitness does not reflect this fact. Worse, with a static fitness some of those may be included as well in the sifted set but only give a very small contribution to diagnosis. If many such contributions accumulate they may mask the existence of a few spores able to more effectively split the set. Experimental results show that this is indeed the case.

The new sifting process leads to vastly improved results with respect to the previous one: the same diagnostic power is obtained with just a fraction of the spores. Since spores are very short programs they are relatively easy to fault simulate. One drawback of the spore set is that, although very large, it does not include all opcodes and all operand combinations, even for opcodes present in the set. This is a consequence of their generation process: it starts from an existing set of test programs, and can only extract information contained therein, but does not generate more.

This leads to a simple idea for evolutionary improvement of the spore set: generate more spores by mutation of existing ones. Every spore has a fixed structure, composed of initialization of the processor state, execution of a single instruction, called target instruction, and observation of the results. The mutation is not intended to change this scheme, but rather to work within it.

The performed modifications are of two kinds: small changes of the operands related to the target instruction, and arbitrary changes of executed operation. An operand can be incremented or decremented by 1 or 2; it may undergo a single bit flip; the opcode can be changed with a different one having the same encoding length and addressing mode of the one replaced. The reason for this different treatment of operands and opcodes is that the cardinality of the opcode set is much lower than the cardinality of the operands space.

The evolutionary process starts from the basic set and generates one new spore from a randomly chosen one.

This is first compared with the existing ones to discover if it is identical to one of them: if it is, there is no need to evaluate it. To evaluate the spore a fault simulation is performed, collecting the signature for each fault, after which the equivalence class number is recomputed. If the new spore is able to split some fault subset then it is retained in the final set, together with simulation data; otherwise it is still retained as an existing (but useless) spore, and its simulation data are discarded. In this way it is ensured that every generated spore is only evaluated once.

To experimentally assess the effectiveness of the modified method, the same case study presented above has been tackled, allowing to perform a systematic comparison at each step of the proposed workflow. All the experiments were performed in a SUN Blade processor-based workstation.

In the case of T_M the sporing process generates about 60,000 spores. This huge number of spores in sDT_M is mainly due to the loop-based code organization of T_M. On the other hand, by applying the sporing process on T_A, 406 spores are generated, since T_A does not contain loops. Those numbers reflect in the execution time.

Spore characteristics have to be separately considered for executive and branch spores. Spores in first category are composed in average of 10 instructions and their execution has duration of 140 clock cycles; in terms of memory occupation, an executive spore ranges between 18 byte and 22 byte. In the branch scenario, spores are up to 21 instructions long and requires for up to about 180 clock cycles to be executed; a branch spore may occupy up to about 50 bytes.

The methodology prescribes that the spores set should be fault simulated. Since this step had not undergone any change, it has not been repeated, and the existing coverage matrix has been reused.

Spore set	**Spores**	**Application time**	**Size**	**FC**	**dr**
sDT_M	60k	8.7M	1.5MiB	92.52%	97.96%
sDT_A	771	112.3k	19.4kiB	90.06%	95.87%

Table 3.4. Results of the sporing process

Table 3.5 summarizes the main aspects of the initial spores sets obtained in both cases. The diagnosis application time of the sets is measured in clock cycles. The *dr* values for sDT_M and sDT_A obtained during this phase will remain unchanged until the evolutionary improvement phase.

In the case of T_M, the sporing process generates about 60,000 spores. This huge number of spores in sDT_M is mainly due to the loop-based code organization of T_M. On the other hand, by applying the sporing process on T_A, 406 spores are generated, since T_A does not contain loops. Those numbers reflect in the execution time.

Reduction in the number of spores belonging to sDT_M and sDT_A is performed by the new sifting process. It is worthwhile to note that the modified sifting guarantees that the diagnostic properties of the spore set are preserved intact in the basic set. Indeed, spores in the spore diagnosis set are discarded from the basic set if they do not contribute useful diagnostic information. They are, however, all considered for inclusion in the basic set.

Basic set	**spores**	**Application time**	**Size**	**FC**	**dr**
bDT_M	449	146.7k	13.2kiB	92.52%	97.96%
bDT_A	261	78.0k	6.5kiB	90.06%	95.87%

Table 3.5. Results of the sifting process using the coarse classification

Table IV presents the information concerning bDT_M and bDT_A, obtained by heuristically sifting sDT_M and sDT_A. bDT_M presents a significantly smaller cardinality with respect to sDT_M; bDT_M is composed of 449 test programs, corresponding to about 1% of the spore set. This result is due to the high redundancy of sDT_M and implies also a considerable reduction in both execution time and memory requirements for the reduced spore set.

The cardinality of the bDT_A is also smaller than that of sDT_A, but not so much if compared with the other sets. The reason for this result lies in the generation methodology for T_A; in fact, T_A does not present so high fault coverage redundancy, therefore sDT_A.

Up to this point, diagnostic ability evaluation relied on pass/fail information, only. *dr* values shown in table IV are obtained by performing coarse classification procedures.

To further increase the diagnostic capability of bDT_M and bDT_A, additional spores are included in the basic sets by means of the incremental evolutionary improvement detailed above.

This process starts with a fine classification process, performed on bDT_M and bDT_A. The results of this preliminary operation are shown in table 3.6. Obtained fine equivalent fault classes whose cardinality is larger than 10 are targeted for extra partitioning. A minimal fault list, only comprising the faults in each class, is built purposely for the improvement process, so that only the faults in the target equivalence class are finely fault simulated.

Basic set	**Spores**	**Application time**	**Size**	**FC**	**dr**
bDT_M	449	146.7k	13.2kiB	92.52%	98.22%
bDT_A	261	78.0k	6.5kiB	90.06%	98.91%

Table 3.6. Results of the sifting process using the fine classification

The mutation process starts from those spores dividing coarse equivalence classes during the fine classification process. The final diagnostic test sets, fDT_M and fDT_A, own a very high diagnostic ability.

Final set	**Spores**	**Application time**	**Size**	**FC**	**dr**
fDT_M	852	239.8k	18.3kiB	92.52%	99.91%
fDT_A	622	190.6k	15.6kiB	90.06%	99.85%

Table 3.7. Results of the improvement process

The results reported in table 3.7 deserve a more detailed explanation. fDT_M is enlarged by 403 new spores after the evolutionary improvement with respect to the bDT_M. On the other hand, the number of programs in fDT_A is 361 more than in bDT_A.

Table 3.8 summarizes the evolution of the diagnostic power $dp(1)$ and $dp(10)$ and the diagnostic resolution *dr* along the generation steps for both initial sets test sets T_M and T_A.

It can be noted that the gap between the $dp(10)$ of fDT_M and fDT_A is of 3.1%. It means that about 650 faults classified in classes smaller that 10 by fDT_M are included in classes greater than 10 by fDT_A. Explanations for this difference have to be sought for in the T_M and T_A characteristics.

First of all, the difference in the fault coverage figures impacts on final results; in fact, 332 faults not covered by T_A are classified in classes faults smaller than 10 when starting from T_M. Moreover, T_M and T_A code organization may affect the generation process. The very large number of spores originated from T_M does not provide a high diagnostic capability in bDT_M, but it provides much more information to the final generation process during the fine classification process. In contrast, the small number of spores originated from T_A confers a better diagnostic capability to bDT_A, but provides little information to the final generation phase. In this case, the fine classification process mainly partitions small classes while the greatest classes are less touched.

	$\boldsymbol{sDT_M}$	$\boldsymbol{bDT_M}$	$\boldsymbol{fDT_M}$	$\boldsymbol{sDT_A}$	$\boldsymbol{bDT_A}$	$\boldsymbol{fDT_A}$
$dp(1)$	33.2%	47.0%	70.6%	26.2%	49.6%	61.8%
$dp(10)$	54.7%	69.2%	87.0%	50.0%	76.3%	83.9%
dr	97.96%	98.22%	99.91%	95.87%	98.91%	99.85%

Table 3.8. Changes in the diagnostic power along with the generation steps

More precisely, tables 3.9 and 3.10 report in detail the diagnostic results during the generation process for both manufacturing test sets. The rows refer to the size of the equivalence classes, and the columns report the number of faults belonging to classes of that size for a given diagnosis set. Only the sets obtained after sporification are considered, as the initial test set is supposedly not suitable for diagnosis. Only faults covered by the diagnosis sets are reported in this table, whereas uncovered faults are neglected.

EC size	$\boldsymbol{sDT_M}$	$\boldsymbol{bDT_M}$	$\boldsymbol{fDT_M}$
1	4,017	5,686	8,542
2	948	1,034	886
3	630	675	444
4	260	312	244
5	185	225	130
6	132	96	90
7	189	98	84
8	104	96	56
9	99	81	36
10	60	70	10
11 - 100	1,403	1,184	1,271
>100	4,073	2,543	307

Table 3.9. equivalence class summary for generation starting from T_M

EC size	sDT_A	bDT_A	fDT_A
1	3,084	5,845	7,275
2	868	1,360	1,240
3	621	651	501
4	460	536	424
5	200	260	200
6	174	138	108
7	140	84	56
8	144	56	40
9	90	36	18
10	110	20	10
11 - 100	1,660	373	1,315
>100	4,227	2,419	591

Table 3.10. equivalence class summary for generation starting from T_A

Costs for the discussed methodology should be considered from two distinct points of view. The former concerns the generation costs, e.g., fault simulations, heuristics computations, classification processes. The latter involves the application costs on suitable ATEs such as the amount of memory or the application time.

As a measure of generation costs, both the experiments shown in this section have been executed on a SUN Blade I workstation with 1GB of RAM.

In general terms, major generation costs are fault simulation processes. Initially, pass/fail fault simulations are required to proceed to the sifting phase; the fault simulation process of the spore sets sDT_M and sDT_A asked for about 225h and 3h, respectively. The fault simulation process is considerably lighter for the spore set generated from the evolutionary test set due to its smaller cardinality.

The sifting process, operating on the stored pass/fail information, is a minor cost that requires no more than a few CPU hours. Similarly, few minutes are required for the consequent coarse classification process consisting in the pass/fail tree construction. This process occupied about 2.2h to generate bDT_M, and a few minutes for bDT_A.

A reduced number of fine fault simulations complete the preliminary diagnostic evaluation of bDT_M and bDT_A. Respectively, the fine classification processes, including fine fault simulations and output-based tree construction, required about 6 and 3.5 hours to compute the diagnostic level the evolutionary improvement starts from.

The evolutionary improvement consisted in evaluating 8,107 and 3,157 new spores to obtain fDT_M and fDT_A. The time required to improve bDT_A, 129h, is more than twice the time required to improve bDTM, 51h. This difference is due to the better diagnostic resolution initially own by bDT_A. Table 3.11 summarizes the generation costs in both the considered scenarios.

From the point of view of application costs, the execution of fDT_M on an ATE lasts for about 18.3k clock cycles. Supposing a 50 MHz application frequency, this means 4.8 ms. This cost for fDT_A, which is 15.6k clock cycles long, is 3.8 ms.

Generation phase	T_M	T_A
Sporing	0.3h	0.1h
Coarse FS	225h	3h
Sifting	2.2h	0.02h
Fine FS	6h	3.5h
Ev. improvement	129h	51h
Total	362.5h	57.6h

Table 3.11. Times comparison for test set T_M

Regarding the created Fault Dictionaries, the size required for storing the Fault Dictionary created for the fDT_M is about 1.3 MB, while it is about 1.1 MB for the fDT_A generated by the evolutionary approach.

It is interesting to compare directly the results of the two methodologies, to better assess the differences between them. Table 3.12 shows the overall comparison in the final results between the former methodology and the new one. Since the former methodology has only been applied to the manually written test set, only T_M is considered in the following.

	Old Method	**New method**
Programs	7,266	852
Final set size	177kiB	18.47kiB
Application time (cl. cycles)	2,000.0k	246.7k
$dp(1)$	61.39%	70.60%
$dp(10)$	84.30%	84.31%

Table 3.12. Results comparison for test set T_M

The new results compare very favorably with the old ones: the diagnostic power increases by about 9% and the percentage of correctly classified faults remains almost the same. These results are obtained with a diagnostic set much smaller than previously, both in terms of program number, memory occupation and application time. All of these figures decrease by about ten times.

Table 3.13 presents a comparison in terms of required time to perform both strategies.

The total generation time decreases by about 30%. Since both the sporing process and the fault simulation needed for the coarse classification (Coarse FS) remain unchanged, so do the related times. The sifting process is slower, but amounts to a small fraction of the total. The basic test set is much smaller with the new methodology, so the fault simulation performed for the fine classification (Fine FS) is much faster. Finally, the times needed for evolutionary improvement are roughly similar. It should be noticed that a trade-off exists between the duration of the evolutionary improvement and the quality of the obtained results.

For the sake of comparison, two additional experiments have been performed in order to

	Old Method	**New method**
Sporing	0.3h	0.3h
Coarse FS	225h	225h
Sifting	0.5h	2.24h
Fine FS	100h	6h
Ev. improvement	168h	129h
Total	493.8h	362.54h

Table 3.13. Times comparison for test set T_M

assess the suitability of the new algorithm improvements. Thus, mixed experiments were launched using in the first case the old sifting or static sifting, and the new evolutionary improvement method, and in the second case the new sifting or dynamic sifting, and the old evolutionary improvement tool. Table 3.14 presents the values obtained mixing old and new approaches. The results are reported in terms of the diagnostic capability reached by the different sets of diagnostic programs.

	$\boldsymbol{dp(1)}$	$\boldsymbol{dp(10)}$
Static sifting alone	35.70%	58.02%
Static sifting + old EI	61.39%	84.30%
Static sifting + new EI	60.37%	69.42%
Dynamic sifting alone	47.31%	69.20%
Dynamic sifting + old EI	51.50%	73.71%
Dynamic sifting + new EI	70.61%	83.41%

Table 3.14. Mixed experiments

This table shows that the modified methodology allows reaching results with the same diagnostic properties of the first one, but saving significant resources. It also shows that the two halves of the methodology are not independent from each other. The type of information provided by the sifting phase strongly influences the performance of the subsequent evolutionary improvement.

3.2 Modular Diagnosis

Diagnosis is an important activity not only because it can allow process characterization, but also for product repair by the usage of backup resources.

Locating faulty components in manufactured circuits is one of the major tasks involving industries and EDA producers. The efforts in this sense chase both the generation of effective diagnostic tests and the identification of the faulty components as soon as possible. Tools for this purpose are indispensable for yield improvements as they permit the identification of critical sub-areas, due for example to final chip layout or of systematic

marginalities in library components.

In the memory field, the use of diagnostic information to repair memories resorting to spare resources is common practice. For logic circuits there is no similar accepted solution but the emergence of new design paradigms, such as those based on more regular structures, like structured ASICs, and the availability of far more silicon area than strictly required by the functions to be implemented pave the way for new solutions that embed the capability to enhance yield by reallocating backup resources at the end of the manufacturing process.

Several possible structures have been elaborated for microprocessor repair. These schemes consist in the introduction of modular redundancy inside a single processor or shared with other processors embedded into the same system. One critical aspect, other than the reconfigurable structure organization, is the efficient identification of a faulty module to be repaired without significantly decreasing the performance.

An activity has been undertaken pursuing the automatic generation of software-based diagnosis sets able to isolate failing functional modules inside a microprocessor, possibly to be remapped exploiting the available resources left unused in the chip.

The approach follows in its general lines the methodology for microprocessor diagnosis detailed in section 3.1, but is targeted at discriminating faulty modules instead of single faults.

The rationale for this is simple: in reconfigurable architectures, or generally in any architecture that can be repaired by substitution of a logic resource with a backup one, entire logic modules are exchanged. If diagnosis is used for repair, then, it is not useful to exactly pinpoint the location of a fault. It is, instead, important to isolate a faulty module in order to replace it.

The methodoloogy relies on the manipulation of a post-production test set and its subsequent enhancement adopting evolutionary techniques. This test set, that must guarantee high fault coverage, is divided in shorter test programs, called *spores*, and a subset of these is selected as the basic diagnosis set. The diagnostic ability of this set of programs is then evaluated recurring to a diagnostic tree. The construction of each tree branch ends when a leaf contains only faults belonging to the same module.

The leaves that contain faults belonging to different units after the basic diagnosis set analysis are processed by an evolutionary test program generator (TPG) and new programs are added in order to improve the basic set.

One of the goals of the approach is to obtain smaller diagnosis sets compared to traditional diagnosis sets, since it only concentrates on isolating a faulty module instead of a single gate.

The methodology exploits an existing Infrastructure IP (IIP) whose original purpose was aiding software-based self test (SBST). The IIP is able to provide the processor with the diagnosis code, and to gather and store the compressed results.

The advantages are numerous. The construction of the diagnosis set exploits an existing test set, avoiding part of the efforts in covering hard to detect faults. The process is automated, requiring limited human intervention. Being software based, the diagnosis can be performed at-speed, reducing the application time. Since the process only collects the final results of a diagnosis program, the methodology does not require the use of

high-performance, costly test equipment.

The approach is based on three automated steps, similarly to the activity described in section 3.1: initial diagnosis programs generation, diagnostic tree construction and evolutionary improvement of the diagnosis set.

The approach is fully automated and guarantees to attain almost the same coverage of the stuck-at faults obtained by the original test set. It can be noticed that the data path of the processor is fed with exactly the same input set induced by the original test programs. Consequently, the fault coverage on the functional modules is unaltered. The control part, on the other hand, may not follow exactly the same state sequence, so its fault coverage may vary.

Since each spore corresponds to a single instruction instance in the original test set the sporing process generates a vast number of diagnosis programs, and then it is necessary to select an appropriate diagnosis set by choosing the fittest ones. To cope with this issue, the programs were evaluated by computing this fitness function:

$$F_P = \sum_C H_C Num_C$$

where P denotes the program, C is an equivalence class with respect to the applied spore, and H_C is a pseudo entropy value computed as:

$$H_C = -\sum_{i=1}^{N} p_i \log p_i$$

where p_i is the fraction of faults belonging to the equivalence class affecting module i. Finally, Num_C is the cardinality of the equivalence class.

This function needs some explanation. Since an equivalence class is a set of indistinguishable faults only a statistical measure can be used to characterize it. Intuitively it is desirable for an equivalence class to contain faults from as few modules as possible: if faults only belong to a single module no further discrimination is necessary. During the application of successive spores an equivalence class may be split in two or more; for analysis purposes it can be assumed that this partition happens randomly. If a class contains faults from different modules a random splitting of this class is more likely to generate at least an equivalence class containing faults from only one module if the starting class is already very polarized.

An entropy measure is able to numerically express all these concerns. To make sure that spore evaluation is fair and to keep fault coverage as an objective undetected faults are always considered a single equivalent class affecting all the modules in the processor.

Once the fitness function was applied on all the programs, a rank was performed and the redundant programs were eliminated. In this case the elimination was performed as follows: a cumulative set of equivalence classes, with an associated fitness function, is maintained; each spore is successively evaluated to determine how the cumulative set changes after its application: if the corresponding fitness function decreases the spore is kept, since it either splits some multi-module equivalence class or it detects some additional faults.

This process guarantees to preserve the diagnostic properties of the generated spore set, since the cumulative fitness decreases for any useful splitting performed, and every program that performs such a split is kept.

To evaluate the ability of the diagnosis set in isolating faulty modules, a module oriented diagnostic tree has been constructed. The goal is to develop a diagnosis set corresponding to a diagnostic tree whose leaves only contain faults belonging to a single module.

Once the initial diagnostic tree is built, there is the possibility that some leaves, denoted as *weak leaves*, contain faults belonging to different modules. These correspond to sets of faults that affect more than one module, and are generated by test programs that are still unable to uniquely diagnose a faulty module.

Then, to improve the diagnostic tree, these sets of faults, corresponding to weak leaves, are targeted by an evolutionary process able to generate additional assembly programs. The tool used is the μGPv2. In this case, the fitness for the tool is the same fitness function described above.

The methodology was applied for evaluation to a microprocessor compatible with the Intel i8051. A more detailed description of the processor core can be found in section 3.1. The core is complemented with an Infrastructure IP (IIP) able to deliver the test programs and to collect the results. It includes a 24 bit MISR, connected to the parallel output ports, to compute a signature obtained from each diagnostic program execution. Every different signature obtained from the execution of a test program identifies a fault equivalence class. The faults in a class can affect one or more of the processor modules.

Inside the architecture, 5 modules have been singled out: 3 in the ALU unit (ALU1-3) with about 1,700 faults in each module, one in the control unit (CTRL) including 6,062 faults and another in the decode unit (DEC), with 1,090 faults. Neither the internal memory nor the IIP hardware have been included in the diagnosable modules.

The workflow, has been started choosing a set of post production test programs, whose total fault coverage is about 93% of the 12,462 processor faults. Application of the sporing process generated about 60,000 programs, each 7 instructions long on average.

These spores have been fault simulated computing the signatures for each. This process required about two weeks on 3 SUN Blade processor-based workstations exploiting an in-house developed fault simulator vaguely based on the PROOFS algorithm [17].

The spores have been evaluated according to the fitness function and sorted in ascending order. The subsequent elimination of the redundant programs, performed as outlined above, selected 346 spores, composing the basic diagnosis set.

Then the module oriented diagnostic fault tree was built. The post production test set is able to isolate 1,883 single module equivalent classes, including 3,167 faults. The bqasic diagnosis set improves these figures identifying 4,632 single module equivalent classes, or 6,248 faults.

Resorting to the evolutionary approach described above, the diagnosis set has been improved. The percentage of diagnosed faults, that is, faults belonging to leaves that contain faults from only one module, increased by about 25%.

The evolutionary tool took about 6 hours to run, producing an additional set of 120

programs. The final diagnosis set, containing 466 programs, isolates 7,716 classes corresponding to 9,252 faults.

Module number	Test set	Basic set	Final diagnosis set
1	3,167	6,248	9,252
2	1,471	1,082	921
3	820	197	174
4	668	173	171
5	5,464	3,757	1,092
Undetected	902	1,005	852

Table 3.15. Equivalence class summary for the analyzed processor core

Table 3.15 contains a result summary of the fault classification. The first column shows the number of modules in every equivalence class. The columns 2 to 4 report the number of faults contained in each type of equivalence class. For example, the basic diagnosis set isolates 1,082 faults in equivalence classes affecting exactly two modules. It is interesting to note that the total fault coverage decreases by less than 1% after the sporing process; the use of the evolutionary approach further increases the final fault coverage by about 1.5% with respect to the basic diagnosis set.

The final diagnosis set has the greatest diagnostic capability; using its diagnostic results it is in theory possible to repair more than 80% of the faults replacing at most two modules.

In general, gate-oriented fault diagnosis for microprocessor cores requires the use of a large number of diagnosis programs with respect to the total number of isolated faults. On the other hand, module-oriented fault diagnosis is cheaper both in terms of diagnosis set size and of application time.

The methodology could be an effective approach for reconfiguration at the end of the production line.

Chapter 4

System hardening

Many of the considerations about test and diagnosis may be repeated for system hardening. This is true even if it is not a part of the design and manufacturing flow in the same way as test or diagnosis are. Hardening is rather a goal of the design process, than a step or the provider of feedback information for the subsequent design iterations.

Nevertheless, the demand for cheap and effective methodologies is far from satisfied. Even in the case of software techniques, the inconvenience of having to fully control the executable code may add to the time and costs of hardening.

An activity in system hardening has been undertaken with the goal of easing some of the limitations in existing software and hybrid methodologies.

It is based on a technique different from those presented in section 1.3. The methodology belongs to the family of hybrid techniques, but features some differences. First, the additional hardware no longer performs all the checks needed to achieve hardening. These checks are executed by the processor, and the approach exploits its computational power to save hardware resources. Second, it is transparent, in the sense that the application code needs no modification for the approach to work.

In this approach, the system is augmented with a small infrastructure IP (IIP) that sits on the processor-memory bus. This IIP takes care of duplicating all the data processing instructions, and of inserting the corresponding checks. In addition, it recognizes branches in the code, and checks that the basic block switch is correct.

The basic assumptions for the methodology are as follows. The system to protect is assumed to be a SoC, in development phase, inside which a processor is used to run a software application. It is therefore possible to add an IIP core in the system. It is also assumed that the processor either has no cache, or all the cache hierarchy can be disabled.

The SoC can either be implemented on an application specific integrated circuit (ASIC), or inside a field-programmable gate array (FPGA) embedding a processor core. For example, the Xilinx Virtex II Pro contains a PowerPC core, and the Actel M7 proASIC3/E embeds an ARM core.

The IIP monitors each instruction fetch, so it must be connected to the processor-memory bus. In addition, it has to provide duplicated instructions to the processor. This means that the processor-memory bus must be entirely interrupted, and the IIP works as

an interface between the two components. It acts as a memory when seen by the processor, and as the processor if seen by the memory. It has to sit on the memory bus since the processor-cache bus is usually not available for insertion of additional hardware.

It is necessary that the caches are disabled for the approach to work. The IIP intercepts every instruction fetch and replaces every fetched data processing instruction with a complex sequence. If an instruction cache was active, the entire sequence would be stored in it, and the actual following instructions would not be fetched from memory. The problem with data cache is more subtle: data duplication is not employed, so any fault affecting the data cache would propagate to both replicas of instructions using that cache line.

This assumption is consistent with the common practice of disabling the caches to increase the dependability of commercial processors when deployed for critical applications. Indeed, instruction or data caches are seldom equipped with error-correction codes (ECC) or parity, while external memories featuring these dependability options are freely available.

The adopted fault model is the SEU on the processor's memory elements. These include the register file, the status registers and the pipeline boundary registers. To apply the methodology to other fault models it should be extended with more complex replication and check techniques, such as replicate with shifted operand (RESO).

In the developed methodology, the IIP duplicates the instructions the processor has to execute, also inserting instructions for consistency check. The actual check is performed by the processor, and if it fails an error handling routine is called. The IIP itself, however, has to check control flow: every time the processor fetches a new instruction, the IIP checks whether its address is as expected. If not, an error handling procedure is called.

The complete procedure is as follows.

- The processor fetches instruction I.
- The address A of the instruction following I in the program is computed and recorded.
- The state of the processor is saved.
- The instruction I is executed, producing result R_0.
- The state of the processor is restored.
- The instruction I is executed again, producing result R_1.
- The state of the processor is saved.
- The two results R_0 and R_1 are compared, and in the case they differ, an error handling procedure is activated.
- In case R_0 and R_1 match, the state of the processor is restored, and the instruction sequence of the program is re-established by executing a jump to the address A.

As an example, a simple data processing instruction can be considered. Supposing an increment instruction is stored at the address 0, it is replaced by thirteen instructions implementing the instruction duplication part of the methodology.

```
0x0000: add  R2,   1,  R1
```

becomes

```
0x0000: push R4              ;1
0x0004: push R5              ;2
0x0008: mov PSW,  R4         ;3
0x000C: add  R2,   1,  R5    ;4
0x0010: mov  R4, PSW         ;5
0x0014: add  R2,   1,  R1    ;6
0x0018: mov PSW,  R4         ;7
0x001C: cmp  R1,  R5         ;8
0x0020: bne    error         ;9
0x0024: mov  R4, PSW         ;10
0x0028: pop  R5              ;11
0x002C: pop  R4              ;12
0x0030: jmp   0x0004         ;13
```

The original code computes the sum of R2 and 1, storing the result in R1.
The transformed code executes the following operations. At lines 1 and 2 it stores the contents of two registers, on the stack. These registers, R4 and R5, are to be used as a scratchpad for the following operations. R4 is used to save and restore the program status word (PSW), and R5 contains the redundant copy of the result. At line 3 the PSW is stored, then the redundant result is computed. At line 5 the PSW is restored in order to perform the computation starting from the correct conditions. The duplicated data processing instruction, in fact, can modify the processor's flags, so it is essential that their value is preserved. At line 6 the original instruction computes the actual result. At line 7 the PSW is stored again. This time it has the correct value for the computation. It is stored because at line 8 the check is finally performed, necessarily modifying the PSW. If the check fails, that is if R1 and R5 are not equal, the program jumps to an error handling routine, otherwise it proceeds with lines 10 to 13. Here some housekeeping is performed, restoring the PSW and the scratchpad registers. Correct execution is resumed by jumping to the correct address.

The criterion for choosing the two scratchpad registers is simple. They may be any two registers whose content is not critical for interrupt handling routines, and that are not used by the replicated instruction.

It is worthwhile to note that branch instructions, load and store instructions, and compare instructions are not duplicated in this way. Therefore, there is no problem in computing the correct address for resuming execution, or in preserving correctly the PSW. Also, data memory accesses are performed only once.

The hardening of the control flow is performed differently. The IIP implements a simple mechanism to check that instructions are executed according to the expected flow.

Each time the IIP recognizes the fetch of a sequential instruction stored at address A, it computes the next address as $A_{next} = A + offset$, where $offset$ is the size of the fetched instruction. Conversely, when the IIP recognizes the fetch of a branch instruction, it computes two possible addresses, A_{taken} and A_{next} for the next instruction. The first is the address in case the branch is taken, and is computed taking into account the branch type. The second is the next instruction, and is computed exactly as above.

When the next instruction is fetched at address A', the IIP checks whether the program is proceeding along an expected flow. If A' differs from A_{next} in case it just executed a sequential instruction, or if it differs from both A_{next} and A_{taken} for a branch instruction, then an error signal is raised, to indicate that an unexpected address has been fetched.

The effectiveness of the methodology has been assessed experimentally through the use of a test bed. It is composed of a LEON2 processor, a memory, and a software prototype of the IIP. The LEON2 processor executes a benchmark application residing in memory. The test bed has been used both without and with the IIP, in order to compare the results. In both versions of the test bed the instruction cache has been disabled.

The LEON2 is a pipelined microprocessor implementing the IEEE1754 architecture, compatible with the SPARCv8. It is available in the form of a synthesizable VHDL description. The core has been synthesized, instrumented for fault injection and mapped to a FPGA device for speed. More details on the core and the fault injection methodology are reported in section 2.1.

A benchmark suite of three applications has been used:

Matrix computes the product of two 3x3 integer matrices

Ellipf computes an elliptic filter function on a set of 32 samples

Viterbi computes the Viterbi encoding of a stream of 64 symbols

For each version of the test bed, and for each of the three benchmark applications, a fault injection campaign has been performed. To this end, 10,000 randomly chosen SEUs were injected in the register file and in the pipeline boundary registers.

Faults of this kind may be classified in three main categories, depending on the effects they have on the system. *Silent* faults do not affect computation, either because the content of the affected memory element is no longer used, or because it is overwritten before being used again. Faults are classified as *error* if they are detected and elicit an error message or signal. *Wrong answer*, instead, is the label for faults that do affect the computation but are not detected. The name refers to the fact that the results of the computation are not correct, and no indication is given about that.

Clearly, the really critical faults are those in the last category. Indeed, silent faults are an issue only because they might affect system integrity from a physical point of view, but do not concern its logical functioning. In case of an error the obtained results are meaningless, but the elaboration might be repeated to obtain the correct results. In contrast, a wrong answer will propagate through the system without much chance for subsequent detection, possibly affecting the integrity of the mission.

In table 4.1 the number of wrong answers is reported for the test bed without the IIP and with the IIP. The ratio between those is also reported, showing a significant decrease

Bench	No IIP	With IIP	Ratio
Matrix	393	40	9.8
Ellipf	561	76	7.4
Viterbi	12	2	6.0
Average			7.7

Table 4.1. Results of the fault injection campaign

in the rate of wrong answers.

Bench	No IIP	With IIP	Ratio
Matrix	10,758	74,807	6.9
Ellipf	37,987	243,626	6.4
Viterbi	99,971	719,718	7.2
Average			6.8

Table 4.2. Execution times in clock cycles

Table 4.2 reports the execution times for both the non hardened and the hardened system. It can be seen that the execution times increase by a large amount, but on average less than the ratio of wrong answers. This is an important point, since in radiated environments it is assumed that the rate of SEUs is constant, so if a program takes longer to execute it also stands a larger chance of being affected by a fault.

Since not all the faults are detected by the methodology, further investigation has been carried out. On close examination, it is observed that the faults giving a wrong answer are those that modify the execution of banch instructions.

The following code fragment can serve as an example.

```
if (a == b)
  true_branch();
else
  false_branch();
```

Some faults escaping detection are those that change the execution flow in one of two possible ways: either `a != b`, but `true_branch()` is executed, or `a == b`, but `false_branch()` is executed.

It is not possible to detect this kind of error just looking at the sequence of fetch addresses generated by the processor, since they always correspond to valid target instructions for the branch, and to the beginning of a basic block.

The only way to detect these faults is to transmit information about the processor state to the IIP, that should also decode in a more complete way the branches to discover their exact type.

Another possible source of wrong answers are the faults that affect the register file. These faults affect the memory area for both the original and the replicated instruction.

Thus, if a fault affects a source operand before the instruction using it is fetched, it will affect equally both replicas of the instuction, which will compute the same wrong result. It should be noted that even RESO techniques could not cope with this kinds of faults, because the same copy of an operand is used two times. The problem can be solved using real data duplication, but this requires sacrificing the transparency of the approach.

The advantages of the presented methodology with respect to software methodologies or hybrid techniques are several. First, the designer of the hardened system no longer needs to modify the application code. Speed optimizations can therefore be freely used during compilation, and commercial off-the-shelf components are allowed in the system. Moreover, the methodology is totally independent from the application, and can be used as is for hardening any system. Additionally, it saves hardware resources by exploiting the computational power of the system processor. Finally, it introduces no memory overhead, meaning that also memory-tight applications are amenable to hardening with this technique.

It shares its main drawback with software methodologies: the time penalty for application execution is large, amounting to 6.8 times on average for the benchmark suite used.

Chapter 5

Evolutionary Computation

Evolutionary computation is a very large and diverse field. The research community has investigated and is still tackling many facets of the field, from theoretical analysis of the mathematical properties of evolutionary methods to real-world applications, without overlooking the introduction of specialized methodologies.

Differently from the test and diagnosis field, there is still the need for stronger theoretical foundations in the field of EC. Statistical analysis of the evolutionary tools is not trivial, and often the analysis performed on a simple evolutionary scheme is not applicable to its supposedly natural extensions. Sometimes the effort for analyzing a general methodology has led to the creation of different tools, in some respect more abstract than their predecessors.

The difficulty in mathematical analysis has also led to a flowering in applications of EC. A number of general schemes and tools exist, and new applications are proposed at a sustained rate.

During the activity, EC has been approached in two complementary ways: as an instrumental technique for solving CAD problems and perform targeted optimizations, and as a generic optimization technique. The use of a flexible, even if targeted, tool allowed approaching problems for which it had not been thought. On the other hand, this extended usage allowed gaining better insight in the inner operation of the tool, and led to enhancemets and revisions whose effect reflects on its original application field.

5.1 μGP

A large part of the test program generation and diagnosis program generation activities have been performed using the μGP tool. It is a generic evolutionary approach to problem solving with an emphasis on assembly program generation. In fact, its purpose is to generate Turing-complete assembly programs [36].

This means that the tool should be able, at least in theory, to generate programs that compute any *computable* function.

Two main different versions have been used during all the activity. The first, named μGPv2 or simply μGP, has been developed inside the CAD group in the time span between

2001 and 2006. It was derived from a previous prototypical tool with limited capabilities, whose purpose was to instance and reorder a series of code *macros* in the attempt to maximize the fault coverage on a microprocessor. The main application goal of the μGP is therefore test program generation.

The second is named μGP^3, and is a complete reimplementation of the tool. The decision to rewrite it from scratch derives from the realization of the limits of version 2. The general architecture of the tool was preserved, but it has been improved in several ways, making it more general, versatile and, especially, more modifiable and maintainable.

In the following the architecture of the tool is described, along with the various features added. A large part of the description is common between the tools, so they are collectively referred to as μGP. When it is necessary to distinguish between them, reference is made to μGPv2 or μGP^3, respectively.

5.1.1 μGPv2

The μGP is not just a single tool. It is an evolutionary approach to problem solving and optimization. It is composed of three independent but connected blocks: an *evolutionary core*, a *fitness evaluator*, and a *constraints library*. It also comprises some other auxiliary tools, useful for practical application of the approach.

The purpose of the evolutionary core is that of evolving the individuals in a population. To this end it employs several evolutionary operators, grouped in three main categories: *mutation* operators, *recombination* operators and *search* operators.

The external evaluator is in charge of assigning a fitness to every individual generated during the evolutionary process. It often is a script that launches an external simulation tool, feeding it with the individual to evaluate as an input, and collects the results, transforming them in a form usable by the tool.

The constraints library, called *instruction library* in μGPv2, is used to decide the allowed structure for the individuals, and also to map individuals to their external representation for evaluation.

Internally, the μGPv2 represents an individual as a graph made of loosely connected subgraphs. Each subgraph begins and ends with two special vertices, named *prologue* and *epilogue*, respectively. In between these two vertices all the other vertices have a predecessor and a successor vertex. This ordering determines the order of appearance of the vertices in the external representation. Additionally, every vertex can have references to other vertices in the same subgraph, or to the prologue of another subgraph. This representation mimics the appearance of assembly programs. In assembly programs, generally branches are made to other instructions in the same function, or to the initialization code of called functions.

Each vertex can contain a fixed number of parameters. These may be integer, hexadecimal, floating point numbers or symbolic constants. The number and type of allowed parameters and references for each vertex is specified in the instruction library.

All individuals in μGPv2 are contained in a single *panmictic* population. In a panmictic population any individual can exchange genetic material with any other individual. It is therefore not structured geometrically or logically.

The μGP employs a variation of the $(\mu + \lambda)$ strategy. Starting from a population of μ individuals, it applies λ evolutionary operators to produce offspring. Since every operator can produce a different number of descendants, and since every operator may succeed or fail in producing valid individuals, the offspring is of variable size.

The evolutionary process ends when at least one of three possible termination conditions is met. The first is the length of the evolutionary process, measured in generations. After the maximum number of generations has elapsed, the evolution ends. The second is a so-called *steady state*. When a certain number of generation is elapsed without any improvement in the best fitness, then the process is said to have entered steady state, and evolution is ended. The third termination condition is the reaching of a maximum fitness. For some problems the maximum possible value of fitness is known in advance, and can be provided as input to the μGP. When that fitness value is reached the process is terminated, saving computation time.

The initial population may be generated from scratch in a random way, or loaded from an existing population. In both cases, there is the option of generating an initial population of ν individuals, and then bringing it to μ individuals after the first generation. Often $\nu > \mu$ is used, in order to perform a preliminary random exploration of the search space before starting the evolution. If, instead, $\nu < \mu$, the population gets silently filled during the next generations. This second scheme is usable when the fitness function is expected to be both computationally demanding and deceptive.

New individuals are generated from parents by application of *evolutionary operators*. An evolutionary operator in μGP is a non deterministic function that transforms one or two individuals in one, two or more new individuals. Every evolutionary operator has an activation probability, that is the probability that it is used to produce offspring. The probability for a given operator can be different from all others.

Mutation operators in μGPv2 are the following. The *add* mutation inserts a new random vertex in a random point of the individual. The *sub* mutation removes a randomly chosen vertex from the individual. The *add/sub* mutation randomly performs either an add or a sub mutation on the individual. The *modify* mutation changes a single parameter inside a vertex in a totally random way. The *local mutation* changes a single parameter by a random quantity that is small with respect to the entire range of the parameter.

Local mutation deserves a more detailed discussion. The purpose of the local mutation operator is to perform an efficient searching of the nearby solution space around a high fitness individual, thereby enabling faster exploitation of the local optimums. The local mutation operator performs a small change in one of the parameters of the macro. The meaning of "small change" depends on the type of the mutated parameter. For numeric types, decimal integers or float values, it indicates an offset computed with a Gaussian model; for a constants list it is a substitution with the previous or the next constant in the list; for hexadecimal values it is implemented as a bit flip; for references it is treated much like it is for a constants list, so a small mutation on a reference moves it to the previous or the next vertex in the subgraph; for every other parameter type it is ignored.

Recombination operators are two. The *crossover* tries to find two compatible *cores* in the graphs of the two individuals, and exchanges them. A core is defined as a portion of the subgraph in which no vertex refers to other vertices of the same subgraph outside

the core and no vertex is referred by other vertices. More intuitively, a core is a portion of a subgraph that is connected to the rest of the subgraph only by the previous/next relationship between vertices, and not by other references. It should be noted that it is perfectly possible for a core to reference different subgraphs. If this is the case, the referenced subgraphs are moved from an individual to the other without modification. Two cores are compatible if they belong to compatible subgraphs, that is to subgraphs described by the same sections in the constraints. The *safe crossover* is a special implementation of crossover for problems where individuals are either very small or contain a large number of references between vertices. In these cases the cores may span all nodes of the individuals, which get then exchanged and not recombined. The safe crossover works around this limitation by transforming all references in the selected subgraphs in *offsets*. After this transformation, the two subgraphs are cut in two places, and the intermediate sections are exchanged. After exchange, the offsets are transformed back into actual references.

μGPv2 also features a search operator. The *scan mutation* randomly selects a vertex parameter, then generates a new individual for every possible value of the parameter. It silently fails in the case of floating point parameters, since their cardinality would be excessive. The rationale for this kind of operator is that of providing a long-range exploration operator for particularly deceptive, or highly multimodal, fitness functions.

Mutation operators also support the concept of *strength*, indicated with the symbol σ. The value of σ is always in the range $[0 \dots 1]$. In the case of add, sub and add/sub mutations, it represents the probability that the operator is applied again on the same individual after a successful application. More specifically, the operator is always applied once. After that, a random number between 0 and 1 is extracted. If it is less than σ, then the process is repeated, applying the operator and extracting again. Therefore, the probability that the operator is applied only once is $p(1) = \sigma(1-\sigma)$, the probability that it is applied twice is $p(2) = \sigma^2(1-\sigma)$, the probability that it is applied three times is $p(3) = \sigma^3(1-\sigma)$, and so on. The expected number of applications is then

$$\begin{aligned} E &= \sum_{i=0}^{\infty} i\sigma^i(1-\sigma) \\ &= (1-\sigma)\sum_{i=0}^{\infty} i\sigma^i \\ &= (1-\sigma)\sum_{i=0}^{\infty}\sum_{j=i}^{\infty}\sigma^j \\ &= \sum_{i=0}^{\infty}(1-\sigma)\frac{\sigma^i}{1-\sigma} \\ &= \frac{1}{1-\sigma} \end{aligned}$$

In the case of local mutation σ is the fraction of the parameter's range that is used as the standard deviation in the probabilistic choice.

Individuals are chosen for reproduction by means of a *tournament selection*. Whenever

an operator needs an individual to use as parent for new ones, then a competition is held. τ individuals, not necessarily distinct, are selected from the population and compared to each other. The one with the best fitness wins the tournament and is used for reproduction. In μGP τ can be any real number greater than 1. But it is not possible to select only a part of an individual for comparison, so τ is split in two parts: $\tau = \tau_i + \tau_f$, where $\tau_i = \lfloor \tau \rfloor$ and $\tau_f = \tau - \tau_i$. τ_i individuals are always selected, and an another one is added to the tournament with probability τ_f.

If $\tau = 2$ then the tournament selection can be proved to be equivalent to the so-called *linearized-fitness roulette wheel* selection. *Roulette wheel* is the name of a scheme that assigns a selection probability that is proportional to the fitness of the individual. An individual that has a fitness two times higher than another one will also have twice its probability of being selected. It is a popular technique, but it is dependent on the actual value of the fitness, instead of the ranking of the individual in the population. This may be undesirable when a few individuals have very large fitness values, or when there are many with nearly the same fitness. To counter this uncontrolled probabilistic distortion, *fitness linearization* is used. In this scheme, in a population of μ individuals, after sorting them by fitness, the last one will be assigned fitness 1, the second from the bottom assumes fitness 2, and so on, up to the best individual which will have fitness μ. In linearized-fitness roulette wheel selection, thus, the probability of selection grows linearly from the bottom of the population up to the top.

The μGP does not use fitness linearization because in it fitness is always *ordinal*, and not *cardinal*, meaning that it is not the absolute value of the fitness that matters, but its ranking among all the others.

An evolutionary process usually progresses through a first *exploration* phase, followed by an *exploitation* phase. Exploration means that the search space is probed in several distant regions, searching for promising solutions to the problem. As the survival phase eliminates the worst performing individuals, and the best ones are preferentially selected for reproduction, individuals cluster closer together, presumably near an optimum of the fitness function. With the clustering of the solutions the exploitation phase ensues: more and more individuals are generated around the optimum, presumably approaching it as the evolution progresses.

In classical evolutionary schemes, the exploitation phase ends with *convergence* of the individuals around the optimum. Convergence is defined for each gene in the population as the situation where one allele accounts for more than a certain part (usually above 90%) of all the copies of the gene. When all genes have converged, so has the population. There is *premature convergence* when this happens early in the evolutionary process, and all the individuals cluster closely around a local optimum, usually far from the global one and of poor quality. In the μGP the strict definition of convergence does not necessarily apply (see below), but the concepts do.

The evolutionary core employs several mechanisms to preserve genotypic diversity within the population. Diversity is important because it allows to undergo a longer exploration phase, with a better probability of finding good regions in the search space. Even more importantly, diversity is the primary means to escape local optimums and search for better ones when convergence is reached.

The most direct way to preserve diversity is the detection of *clones*, individuals genotypically identical to others already in the population. The μGP allows to scale down their fitness by a given ratio S and possibly keep them in the population, or to set their fitness to zero, effectively exterminating them. When simple scaling is used the first individual in a group of clones retains its fitness f, the second gets a scaled fitness $f_s = fS$, the third is assigned $f_s = fS^2$, and so on. The conditions for the fitness scaling to work are two: first, S must lie in the range $[0 \dots 1]$; second, no individual should ever get a negative fitness, otherwise even clones that are supposed to be exterminated would remain in the population and be ranked better than those individuals.

In order to have a metric of population diversity the μGP computes the total entropy of the population. The computation is performed by considering every vertex inside the individuals, complete with the values of its parameters, as a symbol. Then, starting from the frequency of different symbols, entropy is computed with the classical formula from information theory. The contribution of every individual to population diversity in μGPv2 is computed while the total entropy is computed. The value of the entropy is computed incrementally, adding one individual at a time. The difference between the entropy after the symbols from an individual are added to the total message and the entropy before that addition is the individual's *pseudo delta entropy*.

The computed pseudo delta entropy can be used in the so-called *entropy fitness hole*. A fitness hole is a probabilistic distortion of the selection process. In its original formulation [34] it is a method to control *bloating* in evolutionary processes. Bloating is the uncontrolled, and often undesired, growth in the size of the genotypes of the individuals. The effects of bloating are two: the evolutionary tool occupies an excessive amount of memory, and every fitness evaluation becomes computationally intensive. Bloating occurs because during the evolutionary process some individuals appear, bigger than average, that obtain a marginally better fitness with respect to the others. Most often a large part of the genotype of such individuals is actually useless, but mutations that remove genes from the individual are far more likely to disrupt its structure, and therefore to worsen its fitness, than mutations that increase the genome size.

The original mechanism for the fitness hole is a direct way to counter this phenomenon, modulated probabilistically. With a certain probability p, the selection criterion is not the fitness, but the size. When two individuals are compared, the smaller one wins, regardless of the fitness values. With probability $(1-p)$, instead, the fitness is the selection criterion, as normal. This does not really hamper evolution, because the best individuals always remain in the population, and they can be selected as long as they are not pushed off. In problems where the bloating phenomenon is severe, privileging small individuals may not only save resources, but allow obtaining higher quality results, since evolution can be carried on for a greater number of generations. The name fitness hole comes from the fact that this mechanism represents a hole in the probability distribution for selection.

In the μGP, the fitness hole is used to enhance diversity in the population. With probability e, an entropy measure is used as the selection criterion during tournaments. In the μGPv2 this is the pseudo delta entropy. This mechanism favors the individuals that contribute the most to population diversity.

Older releases of the μGP can only follow a plus strategy, that is a strongly elitist

scheme. Elitism can be seen as the persistence of high fitness individuals in a population. This has two aspects: the duration of this persistence, which in a pure elitist scheme is infinite; and the number of long lasting individuals who belong to the elite (indicated as elite size).

It is stated above that the comma strategy is preferred when the fitness function is deceptive, and the plus strategy can be better when solutions show a complex structure. The μGP features two mechanisms to tune its behavior from a pure plus strategy, as originally developed, to a pure comma strategy. These are *aging* of the individuals and a configurable *elite size*.

In the first mechanism every individual has a defined age, measured as the number of generations elapsed since it first entered the population. After a certain number of generations, individuals are removed from the population. Individuals die not only because of bad performance, but also of old age. Aging allows to calibrate one aspect of elitism, namely the persistence of elite individuals in the population. This enables the user of the tool to choose how much elitism should be incorporated in a single evolutionary process. A pure comma strategy is therefore obtained by setting the maximum age of the individuals to one generation, whereas a pure plus strategy is used when the maximum age of the individuals is infinite, or any number greater than the maximum number of generation for the process.

Comma strategy, completely replacing the population between generations, also discards the best individual obtained during the process. To avoid losing the best solution, two different modifications to the base strategy are possible. The first, minimally intrusive one, is the addition of an auxiliary population. In this population the best individuals obtained during the process are saved, in order to have them available at a later stage. The auxiliary population in usually small, and in the minimal case it contains just a single individual. The second mechanism is the creation of an *elite* inside the population. The elite is the set of the first E individuals, after ranking by fitness. During the replacement phase, the elite is retained, and only the rest of the population is discarded. In this case the individuals in the elite remain available for further evolution. Usually also the elite is kept small, possibly containing a single individual. Choosing the elite size again has the result of tuning the effect of elitism in the evolutionary process. If the elite is very small compared to the total population then the evolutionary process is allowed a greater freedom in the exploration of the search space.

The μGP implements the concept of an elite as a number of individuals that do not age as the generations pass. If a pure plus strategy is used in evolution, the concept loses meaning, but it becomes significant as soon as individuals have a chance to die of old age.

Both mechanisms allow tuning the behavior of the tool from a pure plus strategy to a pure comma strategy with many possible intermediate steps.

The core uses all the parameters described above to direct the evolutionary process and optimize performance. Some of these are provided by the user and never changed, while others are self-adapted to follow the various phases of the evolution. The self-adapted parameters are the operator activation probability, the operator strength, the tournament size. The user can specify an initial value for all these parameters, and can also specify a range for the operator activation probabilities and the tournament size. Self-adapted

parameters are also called *endogenous* parameters, since their value depends on the internal state of the algorithm.

The activation probabilities are adapted based on their success rate. Success is measured classifying the newly generated individuals in five categories, that depend on whether they are better than the best individuals, their parents or the last individual in the population. The success index for every operator is computed as the ratio between the weighted sum of the number of successful individuals and the number of operator applications in the last generation. If an operator is successful, its activation probability increases, and decreases if it is not successful. Since the operator probabilities should add up to 1, they are normalized, and the actual variation may be different from the expected value.

Tournament size is also self-adapted based on success rates. In this case what matters is the global success rate, not that of any single operator. If the current best fitness is improved the tournament size is increased, if all individuals are worse than the last in the population it is decreased, otherwise nothing happens.

Operator strength is adapted exactly as the tournament size. In case of success it is increased, whereas in case of failure it is decreased.

The amount by which every parameter is adapted depends on a further parameter, named *inertia*. Inertia is set by the user, not self-adapted, and should lie in the range $[0 \dots 1]$. For every parameter a new target value is computed, and the actual new value is computed as $P_{new} = \alpha P_{old} + (1 - \alpha) P_{target}$, where α is the value of the inertia. The larger the inertia, the smaller the change in parameters from one generation to the next.

The μGPv2 has been used for a host of different applications, some of which very different from its intended original field of microprocessor test and validation. It has been used for polynomial approximation in the field of antenna array design, for function generation, much like a conventional GP tool, and for generation of *warriors* for the *CoreWar* game. All these activities allowed enhancing the tool, starting from a simpler version than that described here, and adding many of the features detailed above. Over the years, it proved very useful, and was the workhorse for many activities. However, this extended usage also brought to light the limits of the tool.

μGPv2 is able to handle only a single panmictic population. Support for any other configuration would require significant coding effort. A multipopulation run is only obtained by launching the tool several times and then merging the results. An *island model*, that is a configuration comprising several independent populations but where at prescribed intervals individuals, and thus genetic material, can *migrate* from one to the others, is not supported. The fitness is not a general multiobjective fitness, since values are used in a prioritized manner. In general, the most serious limitations were that many modifications required a significant programming effort. The tool, though implemented in a modular manner, was tightly knit together. Modifications and updates layered one on top of the others, and the original structure of the tool became obscured by all these additions.

The decision was then reached to rewrite the tool from scratch, employing sound software engineering concepts, with the goal of obtaining a solid, reliable and easily modifiable tool.

5.1.2 μGP^3

The first activity for the new tool has been the development of its specifications. This step is critical because both performance and computational capabilities depend on it.

The final decision for the functional specifications has been to make it open-ended. The purpose for this is to allow experimenting with different configurations, involving the evolutionary strategy, the structure and number of the populations, and so on. This required flexibility suggested implementing the tool in library form. This means that a lower layer of services useful for evolutionary activity is provided, and the high level layers implement the actual evolutionary algorithm and its configuration.

One of the requirements was that the tool be modifiable with relative ease in several of its internal mechanisms. It should be possible to define new evolutionary operators and possibly modify existing ones to adapt the tool to new application fields.

It is required that the user is able to change the internal structure of the populations. This can be useful to promote differentiation inside the population. A single panmictic population tends to cluster its individuals close to each other in the search space, whereas a geometrically structured population, such as a lattice, can develop subsets of genetically similar individuals isolating them by distance.

The number of populations is also subject to change, since many problems benefit from a multi-population configuration. Mechanisms for the transfer of genetic material from a population to the others should be easy to define, to make actual island models possible.

Another desired point of intervention was the replacement policy. The choice between comma strategy, plus strategy, steady state and possibly others is important to tackle different problems.

Programming language choice constitutes part of the specification: in order to be able to use as flexibly as possible the concepts made available by the library, these are implemented using an object-oriented language. This choice allows, for example, to define new evolutionary operators by simply writing a new class that inherits its main properties from an Operator base class. Since evolutionary methods are typically CPU-intensive the project should be implemented with a language that allows generation of efficient executable code; on the other hand, no particular presentation (graphic or otherwise) capabilities are needed, since the tool is to be used mainly from a command line. This makes C++ the language of choice for implementation.

One of the earliest choices to be made was how the program interacts with the outside world. For portability reasons it has been decided not to make graphic capabilities available, and stick to the command line interface, as in the previous version of the tool. This choice is also coherent with the expected use of the tool as part of a larger problem solving or optimization process. In this way it is easy to use the μGP from inside a script that launches it repeatedly, changing some setting.

The absence of a GUI, and the large number of possible settings for the tool implies that configuration files are extensively used. Portability of these files has been deemed important, so XML with XSLT has been used. The constraint library, the general settings, the settings for every population, and the settings of the log subsystem are all in XML. The use of a standard language allows inspecting the files with graphical tools, such as

web browsers, available on many platforms.

This choice has been extended to the intermediate files produced by μGP^3 at the end of every generation to describe the internal status of the evolutionary algorithm, so these also can be inspected. The internal status of the algorithm includes all the populations, the individuals inside them, their fitness values, their ranking, the statistics for every operator, and also the internal state of the random number generator. This is important to allow resuming a process as if it had never been interrupted.

On the other hand, the use of XML is at least inconvenient for the files containing the individuals, as they have to be syntactically valid for evaluation. In the case of test programs, for instance, the XML description would have to be translated to assembly language.

Lastly, fitness representation has been defined. The solution previously developed for μGPv2 has been preserved, that is to represent the fitness as a sequence of floating point values, all on one line, separated by spaces and optionally followed by a comment string. The number of values is set at the start of each run, and is equal for every individual inside a population for the duration of the run. The number of values can, however, be changed from one run to the next. This format never limited the use of the tool, and is also suitable for multiobjective optimization.

The μGP^3 implements a layered architecture, built on the base concepts of *tagged graph* (TG), *constrained tagged graph* (CTG) and constraints library (CL).

The goal of the layered architecture is to neatly separate base concepts, such as the internal structure of the individuals and the details of the CL, from higher level concepts, more directly related to the evolutionary method. Ideally the base classes will never be modified, but those using them will, to reflect the shifts in underlying concepts.

The TG is the base structure used in the μGP^3, providing the foundation for every operation on the cultivated individuals. The constraints library describes the possible phenotypic content of an individual, as well as its structure. Constraints define the syntactic appearance of *valid* individuals. The application of these constraints turns the tagged graph into a constrained tagged graph.

The concept of individual lies upon the CTG, as a number of CTGs compose an individual. In μGP^3 the Individual class, implementing the corresponding concept, is meant to be the lowest level class subject to user modification. Modifications are foreseen mainly to store additional data inside the individual, such as a hash signature or an entropy measure.

Above the Individual class there is the concept, and corresonding class, of population. This class is also subject to modification, since the activities of selection, mating, reproduction and survival are managed inside it. A population, in addition, is associated with a set of applicable operators.

Finally, the evolutionary algorithm exploits the Population class. This class is one of the most volatile, since it is expected that users of the tool will act mainly at this level to change the operation of the tool.

The tagged graph is conceptually a directed graph, whose non directed version may be connected or unconnected, in which every vertex and every edge owns at least one tag. Every tag is identified with a name, unique inside every vertex or edge, and has a value.

In this way information can be attached to every element of the graph in the most generic possible way. Every element of a tagged graph has a name. Names of the vertices are unique within a population, whereas names of the edges are unique within the set of edges starting from a vertex.

The edges starting from a vertex belong to that vertex, so that when the vertex is removed from a graph they too are removed. In addition every vertex stores a set of back references, to track other vertices that refer to it.

The tagged graph, in itself, could take on any shape or content. In the constrained tagged graph information is stored in a way that corresponds to the indications contained in the constraints library. Every vertex is in direct correspondence with one *macro* in the library, described below. The CTG uses tags to store information whose type is defined in the constraints.

In μGP^3 the CTG must be able to represent an assembly program, so its structure reflects this fact. Every CTG is composed of a number of subgraphs, each of which corresponds to a program subsection or function. Subgraphs are not necessarily connected to the rest of the graph. Every subgraph possesses two special vertices, the prologue and epilogue. Every vertex in the subgraph, except for the prologue and epilogue, has two privileged edges, one going to the next vertex and one going to the previous. In this way the subgraph forms a linear structure, able to match exactly the sequential structure of an assembly program. The edges going to the next and previous vertices are distinguished by the tags that contain the name of the edge. For instance, the edge to the next node will have a tag asserting that the name of the edge is "next". In this way it is easy to distinguish the tags that represent te linear structure of an individual from those that represent explicit references, such as jumps or references to data.

The CTG generalizes the concept of *linear graph*, first introduced in [33].

The individual in μGP^3 is a non empty set of CTG. This distinguishes conceptually the new tool from the μGPv2, since to represent any possible assembly program one CTG is enough. The use of a set of CTG allows representation of arbitrarily complex objects. This feature has been used, for instance, in the evolution of test blocks for peripheral test: the assembly part and the VHDL part of the individual were contained in different CTGs. The dependence of the individual upon the CTG is not modifiable, since all the evolutionary operators rely on this dependance.

It is worth noting that some modification to the Individual class may affect the higher classes, as happens with the introduction of an entropy measure, or of aging. The entopy, in fact, is used to alter the selection statistics inside the population, and the concept of aging directly affects the evolutionary algorithm.

Operators act upon ordered sets of individuals to generate offspring, defined as an unordered set of individuals. This definition generalizes the traditional mutation and recombination operators. Input individuals are ordered to cater for operators that treat them differently depending on their position as parents. On the other hand, offspring is not necessarily ordered since the only operations new individuals must undergo before the next generation are evaluation and ranking, and the order of generation is not significnt for these.

Introduction of new operators or redefinition of existing ones is obtained deriving them

from a base Operator class. Operator application is atomic from the point of view of the evolutionary algorithm, and its outcome does not depend on the past history, but only on the list of input individuals and on the value of the random seed.

The minimalist view for a population considers it just as a set of individuals, a pool on which to operate. In the μGP^3, however, a population is associated with the operators that can be applied to its members, so it is natural to delegate operator activation to the Population class. Also selection of the individuals for reproduction has to be done by taking into account the possibility that the population has some geometrical or logical structure.

In view of these considerations many activities have been delegated to the Population class, even if they are, strictly speaking, part of the EA. Population manages all the details in a generation, including the choice of operators to use based on their past success statistics, the selection of individuals for reproduction, application of operators, aging and survival of the individuals.

Operator activation probability is an endogenous parameter, so for adaptation a statistic of past success is stored for every operator. This is taken care of by the Population class.

The current release of the tool features a base Population class and two derived classes: EnhancedPopulation and MOPopulation. The first implements all the features contained in the μGPv2, including clone scaling and entropy fitness hole, whereas the second is used to perform true multiobjective optimization, using the concepts of Pareto dominance and subdivision of the population into levels. It is necessary to have a separate population class because the use of actual multiobjective optmization changes radically both the survival and the selection criteria.

One noteworthy modification with respect to the previous version is that in μGP^3 the fitness hole uses a real delta entropy measure, instead of a pseudo delta entropy, in both available population types. The entropy of the entire population is computed first. Then, for every individual, the entropy is computed again removing the symbols from that individual from the total message. The difference between the second entropy and the first is the delta entropy of the individual. It exactly measures the contribution of every individual to the total entropy of the population. The delta entropy of an individual can be positive or negative.

Possible modifications to the Population class are several, starting from the population structure. A panmictic population imposes no restriction upon the choice of individuals to recombine, whereas in a lattice-based one the first one may be chosen at will, but subsequent ones have to lie near the first one. Another aspect of parent selection is its criterion: traditionally the most used one is some variation of roulette wheel selection based on fitness, and a simple and successful alternative, used in μGP^3, is tournament selection. It may be useful, however, to use different criteria for parent selection, especially to promote diversity within a single population.

Another possible modification for the population is the addition of a small auxiliary population to store the best individuals generated. Generational strategies, such as comma strategy, avoid premature convergence better than elitist schemes like plus strategy. On the other hand they do not guarantee preservation of the best individuals generated. The

auxiliary population serves as a repository of "officially dead" individuals that preserves the best ones found. From this storage one or more individuals may be extracted and reinserted into the population when evolution seems not to progress. This technique, which can be seen as a variation of elitism, has been employed by practitioners in the evolution of corewar programs.

In the new tool the evolutionary algorithm is a lightweight scheme that heavily relies on the underlying classes and methods for operation. The main duties of the corresponding class are to instantiate the predefined number of populations, associate all or only some of the operators with them, execute the evolutionary steps.

In case an island model is used, Evolutionary Algorithm also has to manage migration of individuals from one population to the others. The delegation of strategy implementation to the Population class also permits to have an island model in which not only the population structure can be different among islands, but there can even be different strategies, like comma and plus, within different islands, without affecting the general EA scheme.

Although not strictly an evolutionary activity, this class also provides a regular dump of the internal state to allow crash recovery.

The constraints library is conceptually separated from the evolutionary concepts and classes outlined above; however, it is extremely important because its purpose is to provide the transformation function from genotype to phenotype. Its implementation in the form of a class that reads an external description makes the tool versatile. The fact that the CL is specified externally is indeed one of the main strengths of the μGP^3 tool.

It provides a synthetic descripton of the possible morphology of the TG, by constraining at the same time the structure and the information that the graph can contain.

A CL is composed of *macros*, each of which can include zero or more parameters. These are referenced inside the CTG by some of the tags associated with every element of the graph. This indeed is what makes the tagged graph constrained: every parameter in every macro has a legal set of values, which can be less than the whole value set for the base type. Every vertex of a CTG is an instance of a macro in the IL, and part of its tags materialize the content of the macro parameters, containing the actual value.

Macros are included in *subsections*, that describe the possible form of the subgraphs of he individual. Every subgraph in the individual is mapped to a subsection in the CL, but several subgraphs may map to the same subsection. Every subsection specifies the minimum and maximum number of vertices allowed in the corresponding subsections, as well as the allowed number of subgraphs that can map to that subsection. Subsections may also be compulsory in an individual. In this way they are generated and dumped in the individual even if they are never referenced. This mechanism is useful to generate, for instance, interrupt handlers.

Subsections, finally, form part of the *sections*, whose purpose is to describe complete objects. Every section in the CL describes the form of one or more CTGs in the individual. Every subsection, every section and the complete library all specify a prologue and an epilogue macro. These are always generated in the individual, and if they are ot needed they may be left empty. This compositional hierarchy in the CL allows representing arbitrarily complex concepts, not limited to assembly programs.

The purpose of the fitness evaluator is the orderly recollection of the evolutionary feedback for the tool. The presence of an implementing class is required to be able to support the parallel fitness evaluation feature. The purpose of this class is to act as a queue of individuals to be evaluated: when the queue is full, or when there are no more individuals to evaluate, all the individuals in the queue are dumped, converting them into actual code, and the external evaluator is run.

The external evaluator may in turn fork several processes, possibly on different machines, taking advantage of available parallelism in computation. This is, however, out of the scope of μGP^3. The only requirement from the tool is that the fitness values are returned in the same order in which the individuals are dumped, since there is no explicit correspondence between individuals and their fitness.

A number of details had to be taken care of when implementing μGP^3. These range from down-to-earth choices to considerations regarding the computational capabilities of the instrument, not just its performance.

It is worthwhile to give some consideration to the concept of equality between individuals: as an evolutionary algorithm exploits three different levels of description for an individual, namely genotype, phenotype and fitness, there are three different ways to compare individuals. Two individuals are genotypically equal if their CTGs are equal: the graphs must be isomorphic and all the tags have to be equivalent. Tags can be equivalent in two ways: tags that contain data have to store the same value; tags that represent references have to refer equivalent elements of the CTG. Even if their genotype is different, two individuals may give rise to equivalent phenotypes; CL macros can describe instruction sequences instead of only single instructions, and it is not necessary that distinct macros describe different instructions, so different macro sequences can be transformed into equivalent code. Fitness comparison is conceptually much simpler, as the only needed activity is the comparison of a series of floating-point values, possibly specifying some tolerance to let the fitness be computed through some measurement. Currently the tool implements the checks for genotypic and fitness equality.

Comparison between fitnesses is also necessary to perform population selection at the end of every evolution step. The concept of inequality between fitnesses changes whether single objective or multi objective optimization is pursued: in the first case a simple lexicographic comparison fits the purpose, whereas in the second dominance is taken into account.

An operator does not need to store an internal state, therefore conceptually all operators should be implemented statically. However, all operators are applied through a method whose name is the same for all: the "operator.generate()" method is necessarily polymorphic. This means that operators have to be instantiated like a normal (stateful) class, but instantiation may be considered an artifact imposed by the language.

An easily overlooked activity is crash recovery: it is essential to confer reliability to lengthy runs, but also comes handy when the user wants to launch a new run starting from the results already obtained. Crash recovery requires rebuilding of all the populations and reloading of the corresponding operator statistics. The Evolutionary Algorithm class is in charge of this, so it is advisable to take particular care when modifying this class.

5.2 Populationless EA

Evolutionary computation is a very wide field, in which different and sometimes counter-intuitive methodologies are proposed. One of these is the use of *populationless* evolutionary algorithms.

Studies on populationless EAs have been undertaken since the first half of the '90s, when several researchers independently proposed the concept [38] [39] [37]. The theoretical or practical reasons for giving up an explicit representation of the population are several, but the resulting tools show striking similarities.

The concept has been explored within the CAD group [40] starting from the gene-centric view of biologic evolution developed by Dawkins [35]. According to the *selfish gene* theory, *evaluation* is performed on the individuals, but the real unit of *selection* is the gene. The genes are considered the real replicators of the reproduction process, as individuals, at least in the world of multicellular organisms, are never copied exactly. Genes are successful if they are able to spread in the population, while individuals act merely as vehicles for their reproduction.

A population, thus, can be seen as a pool of genes, in which the genes themselves spread and reproduce. Individuals, on the other hand, are mortal and their reproduction is not a replication: their genomes are lost with their death like hands of cards as soon as they are dealt.

This view, rapidly absorbed in mainstream biology as a valid complement to the traditional view centered on the individual, has far-reaching consequences. For instance, it implies that the well-being of the individual or its own reproduction are not important as long as it is somehow able to promote the reproduction of its own genes. The selfish gene theory provides insight in otherwise puzzling phenomena, such as altruism and general behavior in social insects.

A new, modular architecture has been developed and implemented for the selfish gene (SG) algorithm, first proposed in [40], to extend its applicability field to mixed integer programming.

In the SG, an entire population is described by its first-order statistics. This means that, for every gene, the frequency distribution of all its alleles is kept. This allows, in principle, to describe an infinite population with a finite amount of information. This population will of course not show an infinite variety of individuals.

Individuals are not explicitly stored in the population, but are generated probabilistically only when they are needed for fitness evaluation, and discarded as soon as the evaluation is performed. The generation of a single individual is equivalent to a random sampling of the described population.

Evolution proceeds through discrete steps: individuals are first extracted from the population, then collated in tournaments and finally the winners' offspring spreads back into the population. This statistically mimics the biological phenomenon of gene propagation in the population.

Since the fitness evaluation depends on the specific application individuals are pushed to compete against each other in pairs. The winner of the competition gets the prize of an increase of its alleles probability at the expense of the alleles of the loser. In this way

it is not necessary to be able to assign a single fitness value to every individual, but it is enough to be able to compare two individuals and determine which one is the winner. This is a property of the SG algorithm that allows to broaden its application scope.

The concept of generation has been introduced in the SG algorithm. As in the standard GA, a fixed number of individuals is evaluated before any update is made on the population. In the SG this means that the random sampling and competition are repeated a fixed number of times before any frequency modification is performed. The frequencies are then modified using the results of all the comparisons performed in the generation.

The generations themselves are repeated a large number of times. Theory holds that initial random fluctuations in the genetic distribution induce a positive feedback phenomenon, in which favorable allele combinations are rewarded over poor ones and become more probable, further increasing the chance of being chosen and rewarded again. In this way the algorithm converges to an optimum of the fitness function. As biological theory suggests, while evaluation is performed at the phenotypic level, selection is performed at the level of genes. And, since the selection process is guided, the linkages between the genes are implicitly enforced, as the famous rower experiment from Dawkins shows.

The Selfish Gene algorithm employs random mutation to maintain diversity and avoid being caught in a local optimum during the evolution process. Since there is not an explicit population the concept of mutation is subtly different from its classical meaning. In this context it means that, for every gene of a generated individual, there is a configurable probability that its contents are chosen at random instead of following the statistical representation of the population.

In the original implementation of SG, an individual is identified by its N genes, each one occupying a fixed locus. Each locus l can be occupied by a certain number of alleles. Alleles are indicated with $a_i^l (i = 1 \dots n_l)$, where l is the locus and n_l is the number of diferent possible values for it. Each allele may appear more or less frequently into individuals composing the population. Let p_i^l be the probability of allele a_i^l in the population.

After each competition, the probability of each allele belonging to the winner is increased by a given amount, while the probability of each allele belonging to the loser is reduced. For the sake of effectiveness, probabilites are stored in the original SG through integer numbers. Let K_i^l be the value stored for allele a_i^l, the corresponding probability is:

$$p_i^l = \frac{K_i^l}{\sum_{j=1}^{n_l} K_j^l}$$

Allele probabilities are modified by adding or subtracting the constant value 1 to the relevant K_i^l . The strength of the feedback is defined indirectly by the value of summation of all coefficient K_i^l.

The approach used in the original SG is intrinsically inappropriate when the set of alleles is not enumerable, like real values over an interval. Since the probability to select a single point in the continuous set of possible values is 0, no number of generations could be enough to sample twice the very same point. This makes it conceptually impossible to obtain a meaningful modification of the frequency distribution of alleles by changing it for single points. This consideration, among others, led to the decision to rewrite the algorithm from scratch, employing a new, modular architecture.

The new framework has been built with an extensible architecture to allow the use of any type of gene in the population representation. The main aim is to clearly separate the high-level evolutionary procedure from the low-level details of the implementation.

The architecture of the framework is hierarchical, centered around the concept of population. The population is composed by a set of loci, each of which contains a gene. Every gene can belong to any type. The frequency distribution of alleles is managed by each gene, with the possibility of a different representation for each type.

Every gene representing an enumerable set is described by a discrete probability distribution, while non enumerable sets are associated with a continuous probability function. Every variable can have its own range of existence, independent of the other ones.

To make the evolutionary process independent from the implementation details the exact mechanism for handling the frequency distribution is further confined inside specific modules. The purpose of this isolation is to be able to change a specific module with a different one without affecting the rest of the framework.

A similar but simpler architecture is employed to handle the explicit individuals: an individual is composed of alleles, each of which may be of a different type, and finally contains a specific value.

A modular architecture also allows to tune the behaviour of the algorithm just by setting a few parameters; these are the strength of the feedback, determined by the total area of the feedback function, and a form of viscosity in the feedback, given by its overall width. The program may approximate a random-mutation hill-climber when the feedback is very strong and a genetic algorithm when a weak feedback is used. A further advantage is that the high-level details of the evolutionary mechanism can be freely tuned; this allows to select the types and number of loci in a simple way, and made it possible to experiment on the concept of generation, as outlined above.

As a preliminary approach to the problem it has been decided to represent real-domain frequency distribution with an equal-interval histogram. Inside an interval the frequency distribution is assumed to be completely flat. This is a coarse representation, but it has been decided to neglect this point on the consideration that it is nearly always possible to tune the number of intervals to cope with the ruggedness of the fitness function. The use of a feedback function $f_f(x)$ makes evolution independent of the number of intervals used.

To obtain a meaningful and effective update of the population while employing continuous-valued genes, in the extended SG the statistical distribution of a locus is modified by adding or subtracting an appropriate feedback function $f_f(x)$. The function is centered in $x' = x_w$ and $x' = x_l$, the actual values of the winner and loser gene, respectively.

The chosen feedback function is

$$f_f(x) = \beta e^{-\beta\sqrt{|x-x'|}}$$

It decreases rapidly near the center point but at the same time propagates the feedback on a broad interval.

α and β are constants over a single run and can be tuned to modify the feedback effect. Since the gene values are defined on a limited interval while the feedback function has an

infinite domain, renormalization of the probability distribution is necessary. If the fitness landscsape is smooth enough this approach is perfectly adequate.

During the experiments it has been noticed that, at various stages of the evolutionary process, random fluctuations of the population's statistical distribution may occur. The reason is simply that individuals are randomly generated based on the genes frequency distribution. In the basic scheme, two random individuals are compared, and the population is updated following this comparison. Just two individuals, however, are a poor statistical sample of a conceptually large population, and a decision to modify a big population based upon such a small sample may lead in the wrong direction.

It has then been decided to increase the size of this sample, extracting a larger number of individuals and obtaining the fitness information for all of them before any update is done. This wider sampling of the population mimics the effect of generations in traditional evolutionary approaches, where numerous individuals compete for selection, and all the offspring undergoes the survival phase. In the prototypical tool this has been done simply by extracting an even number of individuals, comparing them in pairs and storing the result of this comparison for the later update. This is performed exactly as in the simple scheme, but increasing the allele frequency for all the winners and decreasing it for all the losers at the same time. The main point is obtaining a significant statistical sample from a population before updating it.

Compaction of an infinite population into a finite statistical description loses information. In particular, the correlation between the alleles occurring in an explicit population is not recoverable from simple first-order statistics, and infinite possible populations exist that can be represented by a single statistical description. It has been decided to neglect this effect, since the theory tells us that the gene linkages are implicitly enforced by the selection process.

One important advantage of the approach is that when an individual is evaluated and its frequency in the population is altered the same thing is implicitly done on many other neighboring individuals. In fact, if an individual is made more probable by increasing the frequency of all its alleles, then all the individuals that, at least partly, share its alleles also become more probable. This introduces a great deal of implicit parallelism in the evaluation process and consequently in the evolution.

All this makes the Selfish Gene algorithm suitable for a different class of problems with respect to a traditional, population-based evolutionary algorithm. Generally speaking, being able to implicitly represent a very big population, the Selfish Gene allows fast exploration of the research space. Due to the positive feedback that is established in the evolutionary process, moreover, it is also able to perform a fast exploitation of an optimum. Without mutation, however, the algorithm is likely to converge to a local optimum without being able to leave it for a better solution.

A prototypical version of the framework has been developed in ANSI C, and comprises about 3.8k lines of code. Currently two gene types are implemented: enumerable (i.e., integer or boolean) and continuous set of alleles (i.e., bounded real value). Following a competition round the probabilities associated with the various alleles are modified. For integer and boolean genes the alleles of the winners are rewarded and those of the losers are punished by simply transferring a constant amount of probability from the latter to

the former, and no other probabilities are modified. For floating point genes, instead, probabilities are modified over a range larger than a single interval, as defined by the feedback function.

In general for an evolutionary method real function optimization is a difficult task, since the cardinality of the search space is huge. It is therefore interesting to observe the behavior of the proposed algorithm with a varying number of real parameters over the total.

To this end, several experiments have been performed to evaluate the effectiveness of the approach, minimising a function with mixed integer and real parameters. The new Selfish Gene implementation allows doing so in an easy and consistent manner, by simply tuning a compilation parameter.

The function used is:

$$f(\overline{x}) = \sum_{i=0}^{N-1} (0.1 + 0.01i)x_i^2 + \sum_{i=0}^{N-1} (1 - 0.02i) \sin x_i$$

$$x_i \in [-10{,}10] \qquad x_i \in \mathbb{N} \qquad \text{or} \qquad x_i \in \mathbb{R}$$

This function has been chosen as a reasonable compromise between deceptiveness and simplicity. Its global minimum cannot in general be found by a simple hill climber. Any of its parameters can be configured as an integer or real number within a predefined range.

In these experiments the total number N of parameters has been fixed to 10 in order to be able to perform several different experiments.

Real parameters make function optimization harder than integer ones for an evolutionary approach. Convergence on the global minimum becomes slower as the number of real parameters increases. On close examination of the results it has been found that the greater complexity of real parameter optimization also affects the convergence velocity of the integer genes. With 20% real genes the integer genes converge to their final value after less than 44,000 fitness evaluations, while with 50% real genes this happens after nearly 77,000 evaluations. When the real genes are 80% convergence of the integer genes is achieved earlier, but they are very few so the result is not statistically significant.

For comparison, a pure random search has been performed using the same fitness function and mix of integer and real parameters as detailed above.

Random search does not undergo significant performance variations with a varying number of real parameters over the total. This can be expected since a random search does not use any information about the problem structure. It performs significantly worse than the Selfish Gene algorithm, being both unable to reach a comparable fitness level and to approximate the optimum value within the given number of fitness evaluations.

The observation of the evolution of probability distribution for the first real gene is interesting. The initial probability distribution is completely flat. After 33,000 generations the distribution possesses two maximums, a broad one near -1.8, and a narrow peak near 2, caused by the existence of a local minimum near it. As the evolution progresses the secondary peak shrinks, until it ultimately becomes negligible. The actual minimum of the fitness function (about -1.3), meanwhile, becomes ever more probable.

Another interesting issue is the effect of the generation concept on the performance of the algorithm. To see it a few more experiments have been performed, fixing the quota of real parameters. Using the concept of generation allows better convergence of the algorithm in the initial phases of the evolution, but can have a detrimental effect in later stages. A generation size of 10 pairs of individuals has a small effect on the performance, while using 100 changes drastically the behaviour of the tool.

This may be interpreted in a conceptually simple way: in the early stages of the evolution, when the gene pool is still unpolarized, a large generation size allows a statistically significant sampling of the search space, making it possible to update the population in a promising direction; later on, however, the big inertia introduced makes population modification slower, thus hampering performance.

5.3 Local Analysis

In order to gain a better theoretrical understanding of the evolutionary processes used a local analysis has been performed on a simple evolutionary tool. This is the same incremental tool used in 3.1 to enhance the diagnostic power of the program set. The tool, though simple, is peculiar in some of its characteristics, so it deserves a separate discussion.

The goal of the tool is to enhance the diagnostic capability of a set of very simple programs, the *spores*, obtained first through a decomposition phase and then filtering them. The spores follow a fixed and very simple scheme. First the values of the program status word and of the input operands are set, then the feeder istruction is executed, and finally the results are propagated to the processor's ports.

The evolutionary process starts using the reduced set of programs as its starting population. The only allowed operators are mutations, when possible small ones. The operands can be modified by adding or subtracting 1 or 2 to them, or by toggling a single bit. The operator can be changed to a different one with the same length and operands. Mutations can affect one single operand or the operator, but not both.

Recombination is entirely avoided, both to keep the tool simple and to avoid disrupting the fixed structure of a spore. Modification of the structure may have the undesired side effect of making the program dependent on the processor state at its beginning, since initializations may be lost due to the random nature of recombination. Also result propagation may be affected, making the program less effective.

The rationale for using only small mutations is that very similar spores are supposed to cover similar fault sets. A small difference between the covered fault sets directly translates to small equivalence classes (ECs) during diagnosis. Proving the correlation between small spore differences and small differences in coverage directly validates the diagnosis methodology.

The operand mutations can be claimed to be small, since either the arithmetic distance or the hamming distance between parent and offspring is kept small, but mutations in the operator may not have small consequences. Nevertheless, it has been decided to allow them on the basis of two considerations. First, they can actually add value to the diagnosis

process. Mutations in the operator can allow reaching regions in the operand space that would never have been explored changing only the operands for a given operator, but that are traversed for another, compatible one. Second, it is not so trivial to decide what is a small operator mutation. Some imply changing the target processor block, so they may safely be disallowed, but for others the situation is not clear. For simplicity and effectiveness considerations it has been decided to leave all these mutations in the process.

Conceptually, every newly generated spore is fault simulated to obtain the corresponding set of covered faults, even if this is actually done only when needed. The fitness function for each new spore is the number of new equivalence classes produced. A spore is discarded only if its fitness is 0, meaning that it provides no new diagnostic information.

A fitness function defined in this way is dynamic, since it depends on the already existing population. In fact, every spore, taken alone, defines a partition of the fault universe. Whether or not that partition further splits the fault universe depends on the currently existing ECs.

Spores are evaluated only once during the process, and never reevaluated. If they are discarded, they will never enter the population, but once accepted they are certainly retained in the final set. This also means that the order in which spores are produced is significant, since their ability to split older classes in general will decrease.

From an evolutionary point of view, thus, the approach is peculiar and interesting. First, it evolves one individual at a time, but its goal is that of producing a population of diagnostic programs that collectively split the fault universe in as many subsets as possible. It is thus hybrid in nature, as it evolves a population through the addition of single individuals. Indeed, the evolutionary operators manipulate single individuals, and the fitness is computed for every single spore, but its dynamic nature reflects the final goal. At the end of the process the population will not necessarily be minimal. This is not a requirement for the process, and if necessary a final *sifting* phase may be performed. It is also worth noting that, although the goal is the evolution of a population, there is no competition between different populations.

A very low selective pressure is needed for the process, for two reasons. One is that the fitness is dynamic. One implication of a dynamic fitness is that, once computed for a given individual, it progressively loses meaning as other individuals are added to the population. Another is that the first individuals added to the population will probably have a higher fitness than later ones. It would therefore not be reasonable to enforce selection schemes, such as a roulette wheel, based on fitness. The other reason is that the population in this case grows indefinitely, as long as the memory space allows it. No individual is ever discarded from the population, so there is no need for competition among individuals. The simplest available alternative is to entirely suppress selective pressure, giving every individual an equal chance for reproduction.

This also matches well the final goal, which is to evolve a population able to finely split the fault universe, without the need for well-performing individuals. Indeed, whether the definition of best individuals is those that cover the most faults, or those that split the fault universe in the greatest number of ECs, it is not particularly useful to concentrate evolving them. To be diagnosed, faults need to be covered (or *not* covered) by them in different ways from other faults. Concentrating only on a few individuals limits the sets

of covered and uncovered faults to those covered by them plus a small additional set, because their offspring will be very similar to those individuals. The hypothesis, however, is that very similar individuals cover similar fault sets. Additionally, a spore has a low fault coverage, meaning that similar spores will leave a large set of faults uncovered, and therefore undiagnosed.

The main tenet on which the evolutionary process is based, that is, very similar spores cover very similar fault sets, has to be proved. The most direct metric to compare two sets is the hamming distance between their binary representations. Each set associated with a spore is represented by a sequence of bits, one for each fault in the processor; a bit is one if the spore covers the fault, and zero if it does not. The hamming distance between any two such representations gives the number of faults that only one of the spores covers. It does not tell the complete diagnostic properties of the two programs, but gives an upper bound to the size of the smallest EC that they alone can produce.

For diagnosis it is desirable to have small, but not zero, hamming distances, because this means that two programs cover almost, but not exactly, the same programs, and define small ECs.

To perform analysis an initial set of spores has been selected. These are chosen to represent all the possible instructions in the processor's ISA. For every one of them an exhaustive set of mutations has been performed, generating all possible offspring for every member of the initial set. Every new spore has been fault simulated, obtaining the covered fault sets. The main difference with the process described in section 3.1 is that, to keep the analysis as independent as possible from the actual problem, only a coarse fault simulation is performed, gathering just pass/fail information. After the fault simulations the hamming distance between every new coverage and that of the parent spore is computed.

The spores are grouped based on the functional module of the processor they target. For every group the averasge fault coverage, the average hamming distance and the standard deviation of the hamming distance are computed. The results are in general as expected, with some noteworthy points.

As is shown in table 5.1 the average hamming distances are significantly smaller than the average fault coverages for the group. This confirms that a mutated spore generally covers almost the same fault set as the original spore, but not exactly the same.

A very interesting, and important, point is that the standard deviation in the hamming distances for these sets is large, and comparable to the average hamming distance. This means that the distribution is very dispersed, with a large number of different values. Many different values in the distribution imply a good fragmentation of the fault universe. One possible scenario, in fact, is that many mutated spores cover the same fault set as each other, although they cover different faults from the parent one. This possibility must be ruled out if the methodology is to be of any real effectiveness. A similar scenario would imply a small number of different hamming distances in the distribution, represented graphically by a few very high and narrow peaks. In contrast, the actual distribution features a large number of different values, and no single values with very high frequency.

A small but non negligible subset of the offspring features zero hamming distance, meaning that those spores cover the same faults as their parent. These are useless for diagnosis. A large part of the offspring has a little hamming distance from their parent,

presumably providing the best contribution to the diagnostic process. Finally, there is a part of the offspring corresponding to very high hamming distances. These spores may also contribute to diagnosis, but they may generate small ECs only in conjunction with other spores. The origin of these high distances may be the mutation of the opcode in the feeder instruction. This mutation can, as seen above, change the target module, drastically modifying the fault coverage for the original target.

Functional module	**Average fault coverage**	**Average hamming distance**	**Standard deviation**
Multiplier	356	210.11	141.28
Divider	360	118.22	100.05
ALU	604	44.80	54.89

Table 5.1. Results of the analysis

In table 5.1 results are summarized for three different processor modules. It can be seen that the statistical properties of the offspring may change significantly, but there is always a strong correlation between the average hamming distances and their standard deviation. This fundamental property allows achieving good diagnostic results.

The best results are obtained for the ALU. In this case there is the greatest probability that mutation of the opcode does not change the functional module involved, since many operations share the same hardware.

In the end, it can be said that the analysis performed proved the main assumptions behind the simplified evolutionary process described in 3.1 are justified, highlighting at the same time the existence of some small but significant exceptions, whose overall value is not necessarily negative from the diagnostic point of view.

5.4 Games

The games field is seemingly very far from either CAD or evolutionary computation. Indeed, game theory is mainly concerned with the analysis of the mathematical properties of different game types, such as deciding whether a two-player game is an *always winner* or an *always drawer*.

The first phrase denotes games where the first player to act can follow a strategy that will always lead to victory, no matter how the second player counters the actions of the first. The phrase always drawer, in contrast, indicates a game where the second player can always obtain a final tie result. For completeness, the category of *always loser* game should exist, but in this case the first player has always the option of not entering the game in the first place.

Game practice, on the other hand, is more concerned on the actual development of strategies and tactics for a particular game. It may be of little value knowing that a game is an alway winner, in fact, if one has no idea *how* to achieve victory.

One activity that links games with evolutionary computation, then, is the artificial

evolution of strategies for a given game. Depending on the game, a strategy can take on many different forms.

Interesting games offer a large range of possible game scenarios. Ideally an evolved strategy has a fitness that depends on how it fares when confronted with all these scenarios.

The same choice, followed in two slightly different situations, can have very different outcomes. Using chess as an example, changing the position of a single piece on the board can turn a winning move in a disadvantageous one. This suggests that the fitness function for a strategy can be a deceptive one.

For practical reasons it is not generally possible to evaluate a strategy in every possible situation. Even if the set of game situations is in principle finitely enumerable, its cardinality can be simply excessive for actual evaluation using limited computational resources.

If it can be difficult or impossible to evaluate exactly a single game strategy, evaluating all possible strategies may be absolutely prohibitive. In the absence of proven methodologies to *build* these strategies starting from the properties of the game, a player is forced to stick with a *good enough* strategy, without being sure it is the *optimum* one.

In these two characteristics the game field resembles the CAD field. In both cases it may not be simple to build a good solution to the problem using simple techniques. Apparently small changes to a solution can lead to large variations in its final quality. Finally, it may not be feasible to search for an exact solution to the problem. In this case, a trade off is usually struck between computational effort and quality.

Games can then be a good benchmark for an evolutionary tool, even if it has been developed emphasizing its application to test program generation. The advantages of games are two. First, as seen, they provide problems of comparable complexity with respect ot CAD applications, so the enhancements obtained in one field can be applied to the other. Second, games offer a competitive environment to evaluate the methodological improvements. In games solutions are customarily evaluated comparing one against the others, so even small differences in effectiveness can be readily detected.

One particular game, that makes a particularly good match to test program generation, is *corewar* [41]. It is a very peculiar game where two or more programs fight in a virtual computer memory. Programs are written in an assembly-like language called *redcode* and run on a virtual machine named *memory array redcode simulator* (MARS). The memory in MARS, also called core, is organized as a circular array, so that there are no absolute addresses but only relative offsets.

The instructions available in redcode are few, but with a vast array of addressing modes, including immediate, indirect, self-postincrement indirect, and so on. Indeed, it is computationally a rather powerful language. The final goal of a redcode program, however, is not to compute something useful but to win a competiton.

Programs are executed in a time-slicing style, one instruction at a time each. Every program may be composed of different threads; to ensure a fair competition the MARS always gives each program, and not each thread, an equal amount of time. Having multiple threads means being able to perform more operations, but at a slower pace.

A program wins if it causes all processes of the opposing programs to terminate, remaining in sole possession of the machine. This is eventually accomplished by overwriting the opponents' code and making them execute an illegal instruction, either directly or by

jumping to a location containing it.

When a thread executes an illegal instruction it is removed from the execution list; to seriously kill a program all of its threads have to be removed from execution, and this is definitely not an easy task.

In the past years, researchers and amateur players developed impressive programs and subtle strategies, most labeled with evocative names such as scanners, vampires, dwarves, stoners. Redcode programs are commonly called *warriors*, stressing the aggressive nature of the game.

Common strategies to defeat the adversary include laying *bombs* on the core, which means writing illegal instructions at some location; capturing the enemy instruction flow inside useless routines, thus slowing their operation; jumping right inside the other program's code, effectively becoming a second copy of that program. On the other side, to avoid enemy attack many warriors are written small, sometimes giving up some of the flexibility that a longer code allows.

Corewar contests are called *hills*. When a new program is submitted to a hill, it plays G one-on-one games against each of the N other programs currently on the hill. Each warrior gets s_w points for each win and s_t point for each tie. Warriors already present on the hill do not rematch one against each other, but their old scores are recalled. Finally, all programs are ranked from high to low and the last one is pushed off the hill. Thus, as long as a program is present on a hill it can get to the top as the result of a new challenge.

Several hills are currently maintained on the internet. Different hills accept different redcode styles, where the instruction set or program length may change. Games are run with different parameters on different hills. It is customary for corewar servers to offer a choice between different core sizes. The number of matches, maximum number of concurrent warriors or scoring systems may also change.

The dimension of the MARS memory, or core size, profoundly influences all strategies, and is probably the key parameter. The most common core size is $c = 8{,}000$, followed by $c = 8{,}192$, $c = 55{,}400$, and $c = 800$.

The oldest and most famous server is simply named KOTH [43] and still hosts seven hills with different settings. However, the hardest hills at the time of writing are on a server called SAL [44], run by the Department of Mathematical and Statistical Science of University of Alberta, Canada. Differently from other hills, the source code of warriors posted to SAL is not visible to all users, and authors who are not willing to expose their strategies send their latest warriors to this server only, contributing to make the challenge very hard.

All hills with a core size $c = 800$ are called *tiny*, and usually do not accept warriors containing more than 20 instructions. Tiny hills are commonly targeted by evolutionary tools and other automatic optimizers, since the program length allows a certain flexibility while the search space is not huge.

Interestingly, before the tiny hills were introduced, corewar was investigated mainly by humans, writing programs according to strategies set out in advance. However, as happens with other games, such as *go*, changing the space available to the players effectively turns one game into a fairly different one. Strategies devised to play effectively in a big core do not necessarily fare well in a tighter environment.

In theory, a hill with a reduced core is where humans should achieve the best performance, since they can take into account a very small number of independent elements while planning, but have a fairly deep understanding capability. On the other hand, the reduced size of the search space also encourages the use of automatic optimization methodologies.

The tiny hill has been the subject of a previous and separate work in games [42], also with the purpose of eventually enhancing the performance of the evolutionry process in tet program generation.

Nano hills are played by exceptionally short warriors, composed by 5 or less instructions. Contests in this case take place in a reduced memory space of 80 locations. The restrictions in the number of child processes and execution time are also tighter than in the common and tiny hill.

These characteristics make these hills even more attractive for users of evolutionary methodologies. First, the small size leads to a search space that is smaller than that associated with other hills, while still too large to make an exhaustive search practical; this leads to the interesting situation where an automated method to generate the warriors has a chance to perform a significant sampling of the search space, but still needs to use some heuristics to avoid getting lost.

Automated methods are not all the same, however; the usual metric for a corewar warrior is the outcome of its confrontations against other warriors: this not only depends upon the exact composition of the hill, but is also a strongly nonlinear function of the warrior's parameters. Simple hill-climbing does not guarantee to find good results. Evolutionary methods, with their ability to perform both an exploration and an exploitation phase during the search process, can be suited for the task.

The activity has been performed using the μGPv2 tool, and has led to two specific improvements. These are the *safe crossover* and the *scan mutation*.

The tool has been augmented, in a separate activity, with an assimilation tool, able to translate existing test programs into μGPv2 individuals. The purpose of this tool is allowing to further evolve hand written programs, supposed to be already very good. This technique can also be applied successfully to corewar programs.

Recombination is certainly an essential operation in an evolutionary methodology; however, its implementation in early releases of μGPv2 relies on the concept of *graph core* to avoid disrupting the structure of the individuals. The small size of programs leads most of the times to graph cores that are as big as the entire individual. This makes crossover decay in either a swap of the two individuals, which is useless, or a concatenation, which often produces individuals that exceed the 5 instruction limit for the nano hill. The purpose of the safe crossover is being able to cut through the graph cores of the individuals and correctly joining the obtained sections.

Warriors for the nano hill are very small programs, whose functioning depends strictly upon the exact values of all their constants. It makes sense, then, to be able to fine-tune any one of them in the search for an optimum. Even if a local mutation already existed, the strong nonlinearity of the fitness function made a long range search more effective. The scan mutation answered exactly that need, allowing to find the (local) best value for a given parameter, even when the fitness function is very rough.

The fitness function plays a fundamental role in every evolutionary approach. Fitness

must be able to lead the evolution toward the desired goal, or at least away from the less promising region of the search space.

However, due to the peculiar rules of the hills, defining such a fitness function is not easy. Once a certain program has entered the hill, its author can help it by submitting new warriors designed to lose with the first one and struggle reasonably with all the others. Maybe such a warrior is instantly pushed off from the hill, but as a result of its challenge the first program improves its position.

This is a fairly standard practice among expert redcoders and it is considered perfectly acceptable. As mentoned above, the source code of warriors on SAL is not available, and a great amount of expertise is required to exploit such team work between programs.

The problem of devising a fitness function is also hardened by the fact that good repositories of strong warriors for the nano hills do not exist. This lack also affects negatively the assimilation technique.

Four different fitness functions have been implemented for the purpose of the experiments. Three of them are based on the warriors downloaded from the koenigstuhl infinite nano hill [45], whereas the fourth relies on an endogenous approach.

The publicly available warriors compose what is called a *reference hill*. They are a reference because the effectiveness of generated warriors is often measured against this static set of warriors.

The first fitness function, in the following referred to as *fitness A*, simply measured the points earned by the warrior against all programs in the reference hill.

This function can be highly ineffective because, unlike those on the tiny hill, the warriors taken from koenigstuhl infinite nano hill were not competitive, and as a result evolution may be biased. Another source of ineffectiveness in this approach comes from the risk of overspecialization: the search may lead to a warrior that only compares favorably to the warriors in the reference hill, but not against other ones. This risk is common to all approaches that use a reference and lowers as the size, or rather the diversity, of the reference hill increases.

For the second fitness reference warriors were ranked and partitioned into 5 different sets according to their relative strength. The points earned by the warrior against programs in different sets were considered separately, and the 5 contributions were used as terms of strictly decreasing importance for the fitness.

The idea behind this approach is to favor warriors able to compete well with strong warriors. However, the ranking is able to measure only the relative strength. Since these warriors are not a significant sample of the SAL nano hill the approach could be useless if the purpose is entering the latter hill.

In the third approach reference warriors were ranked, and the points earned by the generated warrior against all programs were weighted considering the relative strength of the opponent.

The idea behind this approach is analogous to the previous fitness, as are its drawbacks. However, in this case the distinction between reference warriors is not fixed and an erroneous classification for some of them could be less deleterious.

Further experiments have been performed with a totally different, endogenous approach, referred to in the following as *fitness D*. The process in this case started from

scratch, with 20 random warriors that only serve as a starting point. These initial warriors compose the first reference, and are replaced as the evolutionary process advances.

The evolutionary tool is used to produce warriors that maximize their performance, using fitness A, against these random warriors. The best 20 warriors of the obtained population are then substituted to the existing reference warriors, and the process is iterated until a predetermined timeout.

The use of an endogenous approach allows to avoid overspecialization, but requires a greater computational effort to obtain results, as the warriors have to be coevolved together with their reference.

Four different experiments have been run, using the different fitness functions. The first three experiments used a population of 300 individuals, applying 200 evolutionary operators at each generation. The delta entropy fitness hole was set to 100% to promote diversity. Evolution continued until a steady state was detected, and lasted approximately one day each on a AMD-K7 with 1,024GB of RAM, running Linux.

The fourth, different, experiment has been run using a larger population of 1000 individuals, using 1000 operators per generation, and took about three days to complete.

It is worth noting that some experiments had been performed before the new evolutionary operators were available, but none of them led to a satisfactory warrior. Indeed, none of the obtained programs was even able to enter the hill.

Exploiting the two new operators and the first fitness, the evolutions of warriors follows a distinctive trend. In the early generations the warriors are composed basically of SPL instructions. Such programs replicate themselves into the core (SPL stands for split, and is the instruction for spawning a new process), with no aggressive strategy. Then, some DJN (decrement and jump if zero) instructions appear. Finally, the population is invaded from warriors composed of SPL, MOV (move) and DJN, performing a core clear, i.e., systematically writing illegal instruction on the core.

Warriors evolved using this fitness were all called *Bob*. The first one entered the hill at the 6th position, and later managed reaching the 4th with 155.3 points.

```
;redcode-nano
;name Bob v2.1r1.7408
;author The MicroGP Corewars Collective
org START
START:
mov.i  <-30, $-9
spl.a  #-36, >18
mov.i  >-14, {0
mov.i  >-29, {-2
djn.f  $-2, $-3
```

Interestingly, submitting a newer Bob (Bob v2.1r2.6680) produced the team work mentioned above, pushing the first Bob to the 4th position.

Far more interestingly, although less productively, using the second fitness and the assimilation process, the μGP cultivated a series of warriors named *Onions*.

```
;redcode-nano
;name Crazy Onion I
;author The MicroGP Corewars Collective
org START
START:
spl.f #23, >57
mov.i >-1, {42
mov.i >23, {72
mov.i {40, {-3
mov.i {25, {50
end
```

Crazy Onion I is composed of an SPL and 4 MOV instructions. It tries to cover the core with bombs at the maximum available speed. Since the nano hill parameters allow only 5 child threads, the SPL instruction is critical, and if it is hit the warrior is defeated.

And according to Zul Nadzri, a very expert corewar practitioner and quite a recognized player, Crazy Onion I is almost identical to his Polarization 05, the KOTH of the nano hill at the time Crazy Onion I was submitted. However, no warrior of the Polarization series was assimilated by the μGP since their source code is kept secret by the author. The reason for the large performance gap between the two is the marked dependence on the exact parameter values.

Crazy Onion I was thought to be able to survive long on the hill, but has been subsequently removed.

More interesting results were produced using the third fitness and not exploiting assimilation. Warriors cultivated in these experiments were named from small animals, real or inspired by real living beings. The first one, Paedocypris horridus, is shown below.

```
;redcode-nano
;name Paedocypris horridus
;author The MicroGP Corewars Collective
org START
START:
spl.x #-5, >41
mov.i #37, <2
mov.i {-1, {-2
mov.i >-20, {23
djn.f $-3, <31
end
```

Before submitting it, all other μGP generated warriors were removed to avoid the team work effect. Paedocypris horridus scored 155.9, ranking 2nd on the hill.

It's quite hard to understand why Paedocypris horridus won, and kept on winning. According to corewar experts, it lays a carpet of MOV instruction from 20 locations away which eventually combines with the main program, overwriting the DJN instruction, and

creates a 23 line long warrior (a SPL followed by 22 MOVs). This greatly increases the proportion of time available for bombing with respect to the total. Some of the threads execute the newly created code, resulting in a more effective bombing, and making it more difficult to kill.

The warriors generated using the fourth approach have been named as fancy animals. In about three days of computation, a warrior, named Foggy Maus, has been produced using the fitness D. Its structure, reported below, resembles that of Paedocypris horridus: a split followed by three mov and a djn. However, all constants have different value.

Interestingly, these give the warrior a great versability: Foggy Maus, first entered the hill at the 5th position and has been subsequently pushed to the top of the hill, where it resisted for more than 20 challenges.

```
;redcode-nano
;name Foggy Maus (beta)
;author The MicroGP Corewars Collective
        org start
start:
        spl.a #-35, <35
        mov.i >-24, {-1
        mov.i >-21, <33
        mov.i @-5, {-8
        djn.i $-1, <50
        end
```

Another warrior generated by the μGP, named Muddy Mouse, has been submitted to the SAL nano hill. The couple perfectly illustrates the difference between *winning* against a hill and *remaining* inside it. Indeed, Foggy Maus has ranked higher than Muddy Mouse for a long time. As more warriors have been submitted to the hill, however, the two positions reversed, and finally Foggy Maus was pushed off the hill, while Muddy Mouse resisted and is still, at the time of writing, on the hill.

At the time of writing the two warriors generated using the μGP detain the record for the two longest lived warriors on the SAL nano hill.

Chapter 6

Conclusions

The activity performed has been quite a diversified one. The fields of system test, processor diagnosis, system hardening have been approached with the goal of defining a series of methodologies to solve existing problems. Evolutionary computation, first used as a mere tool to automate the activity, has been subsequently explored as a stand-alone field.

All these fields, apparently disjoint, are actually linked together. The activities of test, diagnosis and hardening all contribute to system reliability. While hardening of a system is one of the design goals, test and diagnosis can be seen as part of the design process itself.

Methodologies developed for test can be used for verifying intermediate design steps, enhancing the confidence that the final product meets its requirements. From the point of view of the user, test guarantees the first part of system reliability, that is its correct functioning.

Diagnosis enters the design process in a different way, providing feedback about the most frequent failure mechanisms. Using that information, designs can be updated to lessen the effect of marginalities in the manufacturing process or in the design procedures. Diagnosis is used to improve production yield, and by reflex reliability.

Hardening is performed to allow using a given system in environments where it could not operate correctly in its basic form, or to decrease the probability of its failure below a target value for safety-critical or mission-critical tasks. The purpose of hardening in many cases is to directly increase reliability.

Evolutionary computation is one of the possible tools to automate these activities. Accomplishment of the goals in system testing, diagnosis, and hardening with manual methods is possible, but it requires a deep knowledge of the system on which these activities are performed, as well as a large amount of labour.

The focus of all CAD activity has been the development and use of software methodologies for the test, diagnosis and hardening of programmable systems, with particular emphasis on microprocessors and microcontrollers. The techniques are meant to be as general as possible, even if they have all been applied to specific systems and refined using them.

In the field of test the activities performed are aimed at several purposes: improvement

of the automatic tools used for test set generation, definition of a general methodology for incoming inspection and stress test of processors for automotive applications, test of peripheral cores inside a SoC, reduction of test application time for a particular class of software methodologies.

The enhancements in the automatic tool have actually been continuous, and trace a path from the first prototype to the latest revision that is much longer than the activity presented above. The end result is that the tool is certainly able to add content to a qualifying test campaign, covering possible use cases that the manual methods do not account for.

A methodology for incoming inspection can only be evaluated from a functional point of view. By definition, the user of a device does not have detailed structural information about it, so no fault coverage can be computed. Nonetheless, the methodology defined takes into account microprocessor features that are now commonplace, but are not considered in more traditional software-based methodologies.

Peripheral test is as important, to ensure correct system operation, as microprocessor test. However, it has been traditionally considered an easy task, not deserving special efforts. But when these peripherals are integrated in a SoC, traditional approaches may become ineffective, due to the reduced accessibility of the peripheral. The proposed methodology uses the computational power of the processor to apply the test to the peripheral core, and is able to reach coverage levels comparable to those obtained on processor cores. Additionally, in its last version, it is mostly automated, reducing human effort to a minimum.

It has been demonstrated that test application time can be reduced for a certain class of test programs without significantly increasing their complexity and size. The general problem of test program compaction, however, is very far from solved, since the adopted methodology is very specialized.

In the field of diagnosis there is a main goal driving all activity. This is the reduction of generation and application costs for a diagnosis set, while improving its effectiveness.

The adopted methodologies all comprise three steps: the decomposition of an initial test set, the filtering of the program set obtained from the decomposition, and an improvement phase using an evolutionary tool.

In one case the goal is diagnosis of single stuck-at faults inside a processor, with the purpose of providing detailed information about the existing faults to the manufacturer. This information is then meant to be used in subsequent design revisions, to improve yield.

In another case the methodology is targeted towards statically reconfigurable architectures. In several applications the size of a silicon die is determined not by the area of the circuit, but by the number of pins. In these cases, backup resources can be put on otherwise useless silicon, and used to repair a faulty device at the end of production. In this case, it is not necessary to diagnose single faults, but only to isolate the module in which a fault lies. The detailed methodology has been demonstrated effective for a large percentage of faults.

The activity in system hardening had the purpose of improving the reliability of a processor-based system, being at the same time as unintrusive as possible. The developed methodology, indeed, is completely transparent with respect to the software development

of the system, and takes up very little resources in hardware terms. Its drawback is that it is not, and could not be, totally effective.

The activity in evolutionary computation has followed various purposes. First, the evolutionary tools have been extensively used to automate the activities of program generation for test and diagnosis. Their usefulness as generic optimization tools has also been probed and enhanced. Generalization of existing instruments for wider applicability was also one of the goals.

Several enhancements have been added to the main evolutionary tool used during all activity, the μGP. New evolutionary operators have been added, features introduced, up to the point where it has been completely reimplemented. The last version is applicable to generic problem solving, and allows using different evolutionary schemes, even during the same run. This added flexibility allowed developing previously unfeasible methodologies.

An existing populationless EA was also reimplemented, to make it appllicable to a larger set of problems with respect to the original version.

Performing a local analysis of an evolutionary process used for diagnosis allowed gaining better understanding of both. It has been possible to prove that the basic assumptions behind the use of a simplified process were true, and therefore that its use was perfectly adequate for the problem.

Finally, a venture in the field of games has been attempted. The results of this activity have been good, and allowed improving the tool in previously unforeseen ways.

In general, the CAD activities provided the original motivation for the use of evolutionary tools, and application of these tools outside of their intended field allowed gaining a better understanding of their functioning, and made possible to add enhancements which reflected positively on test and diagnosis set generation.

Bibliography

[1] Jha, N., Gupta, S., Testing of Digital systems, 2003, Cambridge University Press

[2] Agrawal, V., Bushnell, M., Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits, 2000, Kluwer Academic Publishers

[3] Thatte, S. M., Abraham, J. A., "Test Generation for Microprocessors", IEEE Transactions on Computers, vol. C-29 n.6, June 1980, pp. 429 – 441

[4] Paschalis, A., Gizopoulos, D., Kranitis, N., Psarakis, M., Zorian, Y., "Deterministic software-based self-testing of embedded processor cores", IEEE Design, Automation and Test in Europe, March 2001, pp. 92 – 96

[5] Kranitis, N., Xenoulis, G., Gizopoulos, D., Paschalis, A., Zorian, Y., "Low-cost Software-Based Self-Testing of RISC Processor Cores", IEE Proceedings of Computers and Digital Techniques, vol. 150, issue 5, pp. 355 – 360

[6] Xenoulis, G., Psarakis, M., Gizopoulos, D., Paschalis, A., "Testability Analysis and Scalable Test Generation for High-Speed Floating-Point Units", IEEE Transactions on Computers, vol. 55, issue 11, November 2006, pp. 1449 – 1457

[7] Chen, L., Dey, S., "DEFUSE: A Deterministic Functional Self-Test Methodology for Processors", IEEE Vlsi Test Symposium, 2000, pp. 255 – 262

[8] Parvathala, P., Maneparambil, K., Lindsay, W., "FRITS – a Microprocessor Functional BIST Method", IEEE International Test Conference, 2002, pp. 52 – 58

[9] Fallah, F., Takayama, K., "A New Functional Test Program Generation Methodology", IEEE Proceedings in International Conference on Computer Design, 2001, pp. 76 – 81

[10] Bernardi, P., Rebaudengo, M., Sonza Reorda, M., "Using Infrastructure IPs to support SW-based Self-Test of Processor Cores", IEEE International Workshop on Microprocessor Test and Verification, 2004, pp. 22 – 27

[11] Sánchez, E., Sonza Reorda, M., Squillero, G., "On the Transformation of Manufacturing Test Sets into On-Line Test Sets for Microprocessors", IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2005, pp. 494 – 504

[12] Chen, L., Dey, S., "Software-based diagnosis for processors", IEEE/ACM Design Automation Conference, 2002, pp. 259 – 262

[13] Pomeranz, I., Reddy, S. M., "A diagnostic test generation procedure based on test elimination by vector omission for synchronous sequential circuits", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 19, issue 5, May 2000, pp. 589 – 600

[14] Veneris, A., Chang, R., Abadir, M. S., Amiri, M., "Fault equivalence and diagnostic test generation using ATPG", IEEE International Symposium on Circuits and Systems, vol. 5, May 2004, pp. V-221 – V-224

[15] Boppana, V., Hartanto, I., Fuchs, W. K., "Full fault dictionary storage based on labeled tree encoding", IEEE VLSI Test Symposium, 1996, pp. 174 – 179

[16] Ryan, P. G., Fuchs, W. K., Pomeranz, I., "Fault dictionary compression and equivalence class computation for sequential circuits", IEEE International Conference on Computer-Aided Design, 1993, pp. 508 – 511

[17] Niermann, T., Cheng, W., Patel, J., "PROOFS: A Fast Memory Efficient Sequential Circuit Fault Simulator", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 11, issue 2, February 1992, pp. 198 – 207

[18] Patterson, David A., Hennessy, John L., Computer Architecture - A Quantitative Approach (Second Edition), 1996, Morgan-Kaufmann Publishers

[19] The SPARC Architecture Manual, SPARC International

[20] e200z6 Reference Manual, Freescale Semiconductor Inc.

[21] MPC5554 and MPC5553 Reference Manual, Freescale Semiconductors Inc.

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

[23] Jayaraman, K., Vedula, V. M., Abraham, J. A., "Native Mode Functional Self-test Generation for System-on-Chip", IEEE International Symposium on Quality Electronic Design, 2002, pp. 280 – 285

[24] Chandramouli, R., Pateras, S., "Testing Systems on a Chip", IEEE Spectrum, November 1996, pp. 1081 – 1093

[25] Sánchez, E., Veiras Bolzani, L., Sonza Reorda, M., "A Software-based Methodology for the Generation of Peripheral Test Sets Based on High-level Descriptions", IEEE Symposium on Integrated Circuits and Systems Design, 2007, pp. 348 – 353

[26] Chien-Nan, J. L., Chen-Yi, C., Jing-Yiang, J., Ming-Chih, L. Hsing-Ming, J., "A novel approach for functional coverage measurement in HDL Circuits and Systems", IEEE International Symposium on Circuits and Systems, 2000, pp. 217 – 220

[27] Pomeranz, I., Reddy, L., Reddy, S. M., "COMPACTEST: A Method to Generate Compact Test Sets for Combinational Circuits", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 12, issue 7, July 1993, pp. 1040 – 1049

[28] Pomeranz, I., Reddy, S. M., "Static Test Compaction for Scan-based Designs to Reduce Test Application Time", Asian Test Symposium, 1998, pp. 198 – 203

[29] Oh, N., Shirvani, P. P., McCluskey, E. J., "Control-Flow Checking by Software Signatures", IEEE Transactions on Reliability, vol. 51, issue 2, March 2002, pp. 111 – 122

[30] Huang, K. H., Abraham, J. A., "Algorithm-Based Fault Tolerance for Matrix Operations", IEEE Transactions on Computers, vol. 33, issue 6, June 1984, pp. 518 – 528

[31] Oh, N., Mitra, S., McCluskey, E. J., "ED^{4}I: error detection by diverse data and duplicated instructions", IEEE Transactions on Computers, vol. 51, issue 2, February 2002, pp. 180 – 199

[32] Koza, J., Genetic Programming: on the programming of computers by means of natural selection, 1992, MIT press

[33] Banzhaf, W, Nordin, P., Keller, R. E., Francone, F. D., Genetic Programming - An Introduction: On the Automatic Evolution of Computer programs and its Applications, 1998, Morgan Kaufmann

[34] Poli, R., "A Simple but Theoretically Motivated Method to Control Bloat in Genetic Programming", EuroGP, 2003, pp. 204 – 217

[35] Dawkins, R., The Selfish Gene - new edition, 1989, Oxford University Press

[36] Squillero, G., "MicroGP – An Evolutionary Assembly Program Generator", Journal of Genetic Programming and Evolvable Machines, vol. 6, issue 3, 2005, pp. 247 – 263

[37] Harik, G. R., Lobo, F. G., Goldberg, D. E., "The compact genetic algorithm", Proceedingsof the IEEE World Congress on Computational Intelligence, 1998, pp. 523 – 528

[38] Juels, A., Baluja, S., Sinclair, A., The Equilibrium Genetic Algorithm and The Role of Crossover, Technical Report CMU-CS-94-163, Carnegie Mellon University

[39] Baluja, s., Caruana, R., "Removing the Genetics from the Standard Genetic Algorithm", Proceedings of the 12th Annual Conference on Machine Learning, 1995, pp. 38 – 46

[40] Corno, F., Sonza Reorda, M., Squillero, G., "The Selfish Gene Algorithm: a new Evolutionary Optimization Strategy", Proceedings of the 13th ACM Symposium on Applied Computing, 1998, pp. 349 – 355

[41] Dewdney, A. K., "Computer recreations: In the game called Core War hostile programs engage in a battle of bits", Scientific American, 250(5), 1984, pp. 14 – 22

[42] Corno, F., Sánchez, E., Squillero, G., "Evolving Assembly Programs: How Games Help Microprocessor Validation", IEEE Transactions on Evolutionary Computation, vol. 9, issue 6, December 2005, pp. 695 – 706

[43] http://www.koth.org/

[44] http://sal.math.ualberta.ca/

[45] http://www.ociw.edu/ birk/COREWAR/koenigstuhl.html