Available online at www.sciencedirect.com
SCIENCE DIRECT®
Applied Soft Computing 5 (2005) 315–331
Applied Soft Computing
www.elsevier.com/locate/asoc

Evolutionary software engineering, a review

Timo Mantere[a,*], Jarmo T. Alander[b,1]

[a]Department of Information Technology, Lappeenranta University of Technology, P.O. Box 20, FIN-53851 Lappeenranta, Finland
[b]Department of Electrical Engineering and Industrial Management, University of Vaasa, P.O. Box 700, FIN-65101 Vaasa, Finland

Accepted 11 March 2004

Abstract

In this paper, we review the work applying computational evolutionary methods in software engineering, especially in software testing. Testing is both technically and economically vital for high quality software production. About half of the expenses in software production has been estimated to be due to testing. Much of the testing is done manually or using other labor-intensive methods. To develop efficient, cost effective, and automatic means and tools for software testing is thus highly tempting for software industry. Searching software errors by using evolution based methods like genetic algorithms is one attempt towards these goals.

Software testing is a field, where the gap between the means and needs is exceptionally wide. Despite the great advances in computing during the last 30 years the software development and the testing process in most companies is still very immature, meanwhile the complexity and criticality of the software has increased tremendously.

When testing software, by using any optimization method as a test data generator, we are optimizing the given input according to a selected software metric encoded as a fitness function. The success of genetic algorithms in optimization is based on the so called building block hypothesis. Basically, the genetic algorithms do not find any solitary bug at any higher probability than pure random search. However, evolutionary algorithms adapt to the given problem, in practice this means that a genetic algorithm-based tester generates several parameter combinations that reveal minor bugs and based on this information constructs sequences that will reveal, on the average, more bugs than pure random testing.

Keywords: Genetic algorithms; Software testing; Software engineering; Software production

1. Introduction

Evolutionary algorithms include a branch of evolution inspired heuristic optimization methods, the most well known being: evolution strategies, genetic algorithms, evolutionary algorithms, cultural algorithms, and genetic programming. For further references of evolutionary algorithms see, e.g.

* Corresponding author. Tel.: +358 5 621 2890; fax: +358 5 621 2899.
E-mail addresses: tmantere@lut.fi (T. Mantere), Jarmo.Alander@uwasa.fi (J.T. Alander).
[1] Tel.: +358 6 324 8444; fax: +358 6 324 8467.

1568-4946/$ – see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.asoc.2004.08.004

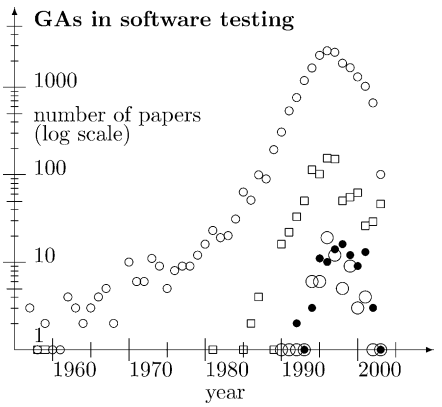

Fig. 1. The number of papers, where GAs are applied to software testing (•, $N = 98$) according to Vaasa GA bibliography database. For comparison total GA papers (○, N = 19,114), GP papers (□, N = 917), and GAs in VLSI testing (◯, $N = 70$). Observe that the last few years are most incomplete in the database. However, it seems that the number of annual GA papers has been settling at about 2000 papers since and including the year 1995.

bibliographies [1–4]. There already exist several international patents [5–8] about using genetic algorithms for software testing, and a number of Ph.D. theses [9–13].

1.1. Genetic algorithms

Genetic algorithms (GA) [14,15] are a group of evolutionary algorithms, that use the evolution principle that was originally proposed by Charles Darwin [16]. Genetic algorithms form a kind of electronic population that mimics the fight for survival, adapting as well as possible to its environment, which is an optimization problem. Surviving and crossbreeding possibilities depend on how well individuals fulfil the target function. GAs are used to solve complex optimization tasks, they do not require the optimized function to be continuous or derivable, or even be a mathematical formula, and that is perhaps the most important factor why they are gaining more and more popularity in practical technical optimization (Fig. 1).

A GA searches the optimum of a problem by calculating fitness values and selecting those individuals that get the best fitness values (selection). The idea of using a GA is self-analytic in the sense that new individuals are created automatically based on fitness values without any human intervention during the test run. However, the need for a manual analysis of test data cannot totally be eliminated, and usually some statistical analysis of test data (results) needs to be done after test runs.

The example in Fig. 2 shows how the crossover and mutation operations, the most common genetic operations, are performed in a real coded GA. In one-point crossover we choose randomly a crossover point, up to which the genes of the Parent 1 are taken to the chromosome of Child 1, while after it the genes of Parent 2 are taken and vice versa for Child 2. In Section 1.8, we will use a more natural uniform crossover, where each gene is randomly taken alternately from either parent. The mutation operator randomly changes the value of a randomly selected gene of the offspring.

Fig. 3 shows how the genetic algorithm operates. At first we create an initial population, e.g. by a random number generator or by supplying known good solutions. The initial population is then evaluated against the fitness function. At this point we usually test the termination condition: we found a solution that is fit enough or a given number of trials has been evaluated. If the termination condition is not satisfied we continue iteration by first selecting new

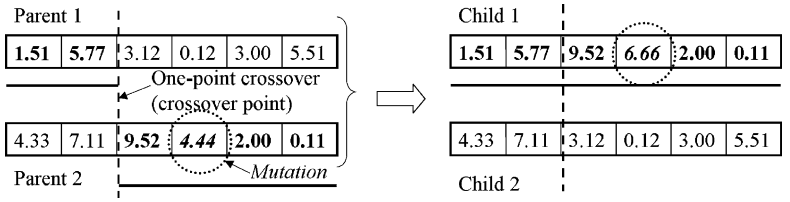

Fig. 2. The one-point crossover and mutation operations applied to Parent 1 and Parent 2 chromosomes resulting new trials Child 1 and Child 2. The genes of Child 1 are shown in bold. The mutated gene is shown in italics.

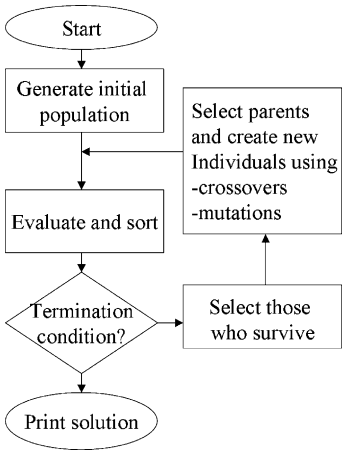

Fig. 3. A genetic algorithm.

parents and those who survive and then generating new offspring by crossover and mutation operations. The new individuals are then tested by the fitness function.

1.2. Other evolutionary methods

Another evolutionary algorithm, Genetic Programming (GP, [17]), is directly related to the evolutionary software engineering. It is one of the few methods to generate code automatically guided only by given criteria, fitness function, that the resulting routine should fulfil. The fitness function gives the distance to the solution. A GP system generates trial solutions, which are then tested against the criteria, the better the criteria is met, the better the fitness. In this review, we will omit automatic code generation. The interested reader is referred to, e.g. bibliography [2] for further references to GP.

1.3. Software testing

The quality of a software system is primarily determined by the quality of the software process that produced it. Likewise, the quality and effectiveness of the software testing process are mainly determined by the quality of the test processes used. More than half of the errors are usually introduced in the requirements phase. The cost of errors is the lower the earlier they are detected. An effective test program prevents the migration of errors between development phases. It has been estimated that an error caught during the system specification phase may be about 50 times cheaper to repair than an error not detected until in the system testing phase [18]. Boehm [19] reports that 12% of the errors discovered in a software system over a three-year period were due to errors in the original system requirements. In practice, we often do not have the mechanism to detect these errors in place until much later—often not until in the function and system test phase [20].

Software errors are human errors, software is written by man, and indeed man does make mistakes. In every commercial software some errors are always present. Some errors are more harmful, visible or costly than others, and testing may never be able to reveal all of them. It seems that software errors cannot be totally prevented, so the best we can do is to try to locate them as early as possible, and at least find and fix the most harmful ones.

The definition of ‘testing’ according to IEEE/ANSI standard [21] is: ‘‘The process of operating a system or component under specified conditions, observing or recording the results, and making evaluation of some aspect of the system or component’’. The definition of ‘software testing’ according to IEEE/ANSI standard [22] is: ‘‘The process of analyzing a software item to detect the difference between existing and required conditions and evaluate the features of the software items’’. ‘Software quality’ means the degree to which a software product conforms to both explicit and implicit requirements. There are excellent online glossaries that explains common terms used in software testing [23] and software engineering [24], and a glossary of terms used in evolutionary algorithms [25].

1.4. Embedded systems

In these days, more and more new products contain microprocessors. The programs of these embedded systems grow and become more complex, causing serious quality problems. A lot of important programming work in Finland during these days is related to embedded electronic devices—mobile phones. Performance is also an important quality and competitiveness factor of modern computer systems.

1.5. History of software engineering

In the early days of software development, testing was regarded as ''debugging'', or fixing the known bugs in the software, and was usually performed by the developers themselves. There was rarely any specific resources dedicated for testing [20]. References to program testing can be traced back to 1950 [26]. In the 1950's even the advanced software systems had very limited interaction, if any, with other systems [27]. Automatic test equipments (ATE) date back to the mid-1950's, when the maintenance of U.S. military electronics started to face formidable problems due to complexity. The solution was the concept of multi-purpose ATE, which promised testing at computer speeds, fully automatic operation by less-skilled operators, the virtual elimination of maintenance documents, and universal designs adaptable to any test problem through the flexibility of programming [28].

By 1957, software testing was distinguished from debugging and became regarded as detecting the bugs in the software [29]. But the testing was still an after-development activity, the underlying objective was to show that the given product worked—and then ship it to the customer. The researchers of computing science did not talk much about testing either. Computer science curricula dealt with numerical methods and algorithm development, but not with software engineering or testing [20].

The term ''software engineering'' was invented in the late 1960's. At that time there was a ''software crisis'', software being expensive, bug ridden, and impossible to maintain [30]. By the 1970's the term software engineering was used more often, thought there was little consensus as to what it really meant [20]. The first formal conference on testing was held at the University of North Carolina in 1972 [29].

By the early 1980s ''quality'' became the popular theme in industry. Software development and testers started to get together to talk about software engineering and testing. Groups were formed to eventually create the many standards we have today (IEEE, ANSI, ISO). International standards are currently becoming too weighty to digest in their full-published form for everyday practical purposes. However, they include important guidelines, baseline for contracts and provide invaluable reference [20]. In 1981, Browne and Shaw [31] stated that the software engineering is a technical activity for which we have developed a large set of ad hoc engineering techniques without a corresponding scientific foundation.

Today, only about 10% of the cost of a large computer system lies in the hardware, while it was over 80% in the 1950s [30]. It has been reported that software costs are growing 15% annually, while productivity is increasing at less than 3% [32]. Despite the enormous advances during the last 30 years or so, the software development and testing process in most companies is still very immature. The complexity and criticality of software has become greater. Even many well-proven methods are still largely unused in industry today, and the development of software systems remains inordinately expensive [20].

1.6. Applying GAs for software testing

Fig. 2 represents the principle of genetic algorithm-based software testing. A GA test generator runs as its own application and communicates with the tested software. The input domain is mediated between the softwares by software interfaces, i.e. with subroutine calls, digital or analog inputs, field bus or Ethernet messages, and message packages. The response might be, e.g. state change in the digital outputs, returned calculation sum, or response message.

The location of the fitness evaluation module (Fig. 4) varies depending on the software metric used. If we are optimizing calculation errors or any metric, that can be directly measured from the response, the fitness evaluation can be combined with the GA application (location 1). However, if we are optimizing, e.g. code coverage that requires code tracing, the fitness evaluation is done at the software side (location 3). If fitness evaluation is done indirectly somewhere between the two routines, e.g. measuring timings in the interfaces, it could be on either side, or it might be somewhere in between the two applications (location 2).

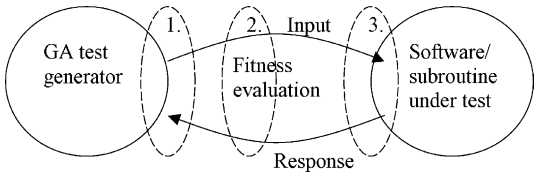

Fig. 4. GA testing system.

1.7. VLSI testing

In the field of automatic test data generation GAs have had great success in VLSI circuit test pattern generation. In VLSI testing the goal is to find the smallest test set that can test the circuit completely. Circuit and software testings differ from each other in that only one program instance needs to be tested while every individual circuit must be tested thoroughly. When the software version is acceptable it can be copied infinitely without a worry of copying errors, i.e. the need of testing each individual. For further references of GAs in VLSI testing see, e.g. bibliography [3].

1.8. A toy example

In software testing the two most commonly studied areas where the genetic algorithms are applied are the test coverage optimization and finding the worst case execution times (WCET). In our own research we have mostly concentrated on the latter one. Our toy example below will demonstrate why it is so tempting to try to find WCET with GAs.

Let us assume that the input domain of a software consists of six parameters p_i, $i = 1, \ldots, 6$, $p_i \in [0, 1]$ and WCET = 6 s, but the allowed maximum time is defined to be 5 s. The traditional way to find WCET could be random or systematic search by testing all the possible combinations with boundary and certain amount of intermediate values, in total n_v values. However, with larger number of parameters n_p this becomes impossible, the number of systematic tests needed n_t increases exponentially as $n_t = n_v^{n_p}$.

Fig. 5 shows how the time delay is influenced by the input parameters. It is easy to see that the assignment $p_1 = 0.0$, $p_2 = 0.2$, $p_3 = 0.4$, $p_4 = 0.6$, $p_5 = 0.8$, and $p_6 = 1.0$ would produce the maximum time delay. However, in the real software several parameters together effect the execution time, and there is also some non-deterministic behavior caused by the state machine nature of programs. If we systematically test all possible parameter combinations when each parameter can have values: 0.00, 0.25, 0.50, 0.75, and 1.00 we have $5^6 = 15{,}625$ possible combinations.

To compare the GA, random, and systematic approaches we solve the toy problem and record the number of parameter combinations used by each method. The GA has population size = 50, mutation rate = 2%, and 50% elitism, i.e. 50% of the best solutions are always selected to the next generation. Table 1 shows how the maximum execution time varied for these three methods. The GA found much longer maximum and average execution times than the random or systematic methods. This difference is caused by the tendency of GAs to favor those parameters that cause high fitness values. As can be seen the random and systematic methods could not find any test case that comes even near the allowed time limit 5 s. Usually there is some safety limit, e.g. so that in the testing the software execution must not exceed 70 or 80% of the allowed absolute limit. In this example both random and systematic search did not reach even 60% of the absolute limit, and thus the software might have passed our test. The GA was the only method able to reach and even exceed the given time limit.

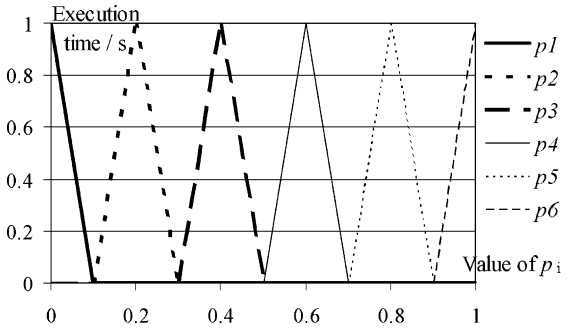

Fig. 5. Toy example: the time delays caused by parameters $p_1, \ldots, p_6$.

Fig. 6 shows the distribution of execution times found by the three methods. All methods generate mostly test cases with execution time zero. Our systematic method was chosen so that it generated only values $i/2$, $i = 0, \ldots, 6$. The GA has clearly found

Table 1
Descriptive statistics of the maximum execution times for systematic, random, and GA methods for our WCET toy example

	Systematic	Random	GA
Maximum	3.00	3.50	5.88
Minimum	0.00	0.00	0.00
Mean	0.60	0.49	1.93
S.D.	0.63	0.54	1.70
N	15,625	15,625	15,625

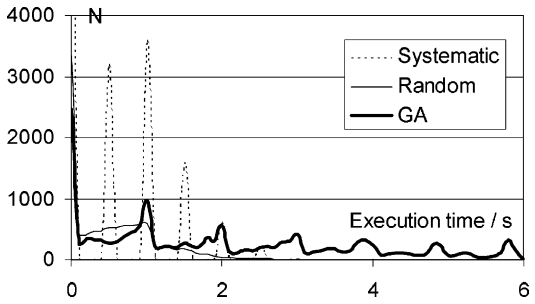

Fig. 6. The distribution of the execution times found by systematic, random, and GA testing.

parameter combinations that cause long response times and have long tail towards the possible global maximum of 6 s.

This example shows clearly how the GA is able to favor those parameter combinations that cause high response values, and this is why it is so tempting to use it for this kind of optimization problems. In this example, we could have found the global maximum easily by statistically analyzing how the parameters effect the execution time with random testing or by being more lucky when choosing the values for the systematic testing. In real life software the dependence between parameter combinations and execution time is not so straightforward as in our toy example, causing often difficulties for statistical analysis or systematic testing. Researchers have assumed that the GAs search ability is caused by its population-based procedure and the building block hypothesis, which claims that the population contains good building blocks and by combining them the GA can generate even better solutions.

Table 2 shows how the fitness of the best individual develops in a GA optimization. From each population only the best individual and the parents of the best solution of the next generation are shown. From the table we can see that in the later generations many individuals share identical gene values. This is a symptom of population convergence. In the early generations the development is more due to crossover (shown in bold), only in the last few generations the mutation (shown in italics) have been the most important operation. In the early generation the best individual is not produced from the best individuals of the previous generation. In the later generations the best individual of the previous generation is usually the parent of the new best individual (see also Fig. 2).

2. Genetic algorithms in software testing

Usually, when using genetic algorithms for function optimization, the GA creates trials, which are then tested by a static fitness function in order to evaluate their fitness as a problem solution. When we are testing software with GAs there is usually not any static fitness function that directly would evaluate the goodness of the trial. Instead GA supplies the test data to the object system under test, which executes some operations according to the received data. The fitness value is evaluated indirectly by observing what the tested system does. The observing can be done, e.g. from the interfaces or by instrumenting the code properly.

When testing software by using a GA as a test data generator we are optimizing the given input according to some criteria. In order to do that, we must define or choose some software metric we are optimizing, and which is measurable from the software, either directly or indirectly. Evolutionary testing can be used with either white box or black box testing techniques. In the white box approach, the optimized metric can be chosen to be, e.g. some test coverage metric; code, condition, or path coverage. There the software execution must be traced, and the aim is to generate a test set that leads to the best coverage. In the black-box approach, there is no need to trace the software execution, but instead to trace what happens in the software interfaces. The optimized metric can be error based, e.g. amount of warnings, calculation or rounding error, leakage of memory, or temporal based, e.g. B/WCET, best or worst execution times or response time.

2.1. Early works

The short history of applying genetic algorithms to the software-testing problems seems to be traced back to the year 1992. The earliest usually referenced paper is Ellis and coworkers [33], in which the testing prototype TAGGER is introduced. The system was used for generating test data for programs written in Pascal.

Table 2
The evolution of fitness in GA optimization

	Fitness value	Param. values					
		1	2	3	4	5	6
Initial population							
Best individual	1.8218	0.7321	0.1077	0.3865	0.0107	0.7880	0.3495
Parent 1	1.3891	**0.1905**	**0.2282**	0.4755	0.8752	**0.8574**	0.0479
Parent 2	0.9424	0.2502	0.7578	**0.9472**	**0.7099**	0.0868	**0.9942**
Second generation							
Best individual	2.0861	0.1905	0.2282	0.9472	0.7099	0.8574	0.9942
Parent 1	1.7841	0.9721	0.4788	**0.6832**	**0.5805**	**0.7979**	**0.7916**
Parent 2	1.5940	**0.0056**	**0.4579**	0.9217	0.2083	0.0595	0.9650
Third generation							
Best individual	2.7282	0.0056	0.4579	0.6832	0.5805	0.79	0.7916
Parent 1	1.8114	0.7321	**0.3493**	**0.3865**	**0.6934**	**0.7880**	**0.3495**
Parent 2	1.5940	**0.0056**	**0.3493**	0.9978	0.0269	0.0239	0.9650
Fourth generation							
Best individual	2.7555	0.0056	0.3493	0.3865	0.6934	0.7880	0.3495
Parent 1	2.6537	**0.0427**	0.7739	**0.2195**	**0.5805**	**0.8428**	0.9703
Parent 2	2.6159	0.0993	**0.2282**	0.3407	0.6884	0.8574	**0.9942**
Fifth generation							
Best individual	3.6104	0.0427	0.2282	0.2195	0.5805	0.8428	0.9942
Parent 1	3.5291	**0.0056**	**0.2761**	**0.4675**	**0.5805**	**0.7880**	0.9335
Parent 2	3.4804	**0.0056**	0.0031	0.6832	**0.5805**	0.7979	**0.9752**
Sixth generation							
Best individual	4.1297	0.0056	0.2761	0.4675	0.5989	0.7880	0.9752
Parent 1	3.9816	**0.0056**	**0.2282**	**0.2195**	**0.5805**	0.8428	**0.9942**
Parent 2	3.5155	**0.0056**	0.8273	**0.2195**	**0.5805**	**0.8176**	**0.9942**
Seventh generation							
Best individual	4.2333	**0.0056**	**0.2282**	**0.2195**	**0.5805**	0.8176	**0.9942**
Parent 2	4.1297	**0.0056**	0.2761	0.4675	0.5989	**0.7880**	0.9752
Eighth generation							
Best individual	4.2899	**0.0056**	**0.2282**	0.2195	**0.5805**	0.7880	**0.9942**
Parent 2	4.1297	**0.0056**	0.2761	**0.4675**	0.5989	0.7880	0.9752
Ninth generation							
Best individual	4.5647	0.0056	0.2282	0.4675	0.5805	*0.7830*	0.9942
Parent 1	4.2899	**0.0056**	**0.2282**	0.2195	**0.5805**	**0.7880**	**0.9942**
Parent 2	4.1297	**0.0056**	0.2761	0.4675	**0.5989**	**0.7880**	0.9752
Tenth generation							
Best individual	5.2079	0.0056	0.2282	*0.3735*	0.5989	0.7880	0.9942

Numbers in bold indicate that this gene is forwarded to the offspring. In the case of the best individual having gene values in italics that gene is not inherited from either parent, but generated by mutation.

Other early papers describing the use of GA for testing behavior based control software for autonomous vehicles, are from Schultz et al. [34–36]. Their goal was to find a minimal set of faults that can be tolerated without significant performance loss of the control system. A chromosome represents a set of initial conditions followed by rules, which specify various fault modes that could be present in the control system. A GA was used to search for potential faults in the software. The object controller software was designed for aircraft and an autonomous underwater vehicle.

The first Ph.D. thesis in the area was by Sthamer [9] who studied the use of GA as a test data generator for

structural testing. The example programs are small procedures written in ADA, including triangle classification, linear search, remainder calculation, and direct sort. Sthamer applies the GA for branch, boundary, and loop testing, and also for mutation testing. He observed that ''GAs show good results in searching the input domain for the required test sets, however, other heuristic methods may be as effective, too''. ''GAs may not be the final answer to the software testing problem, but do provide an effective strategy''.

2.2. Coverage testing

Pei et al. [37] concentrate on pathwise test data generation. By using test data generation by GA they try to define if the selected subpath is feasible or not. They compare their system with Korel [38] and believe that a GA-based system works better because it processes the whole path simultaneously. Pei et al. also claim that their system is superior when compared with many of the commonly used methods.

Watkins [39] deals with path coverage optimization by a GA, using the popular triangle classification problem as an example. The GA reached the same coverage as the random method while sampling a smaller percentage of the complete search space.

Roper et al. [40] have also studied optimization by a GA, trying to find a set of data that will test the program until the required level of coverage is met. Roper [41] has also stated that ''by using a GA one often neatly avoids many of the problems of automatic test data generation encountered by other methods''.

Smith and Fogarty [42] studied test coverage optimization by a hybrid version of a GA and hill-climbing local search. Their application was also the triangle classification problem. They claim that their system can generate test sets that fully satisfy the given metric and reduce the size of evolved test sets. Smith et al. [43] continue that work, by generating with a GA test programs for verifying the design of a microprocessor. The test problem is a VHDL model of hardware, the test coverage is optimized and the results are compared against the random method. The distribution of the results for a GA is significantly more skewed towards higher fitness values than for the random method.

Warfield [5] has received a United States patent for an automatic software testing tool that generates test scripts based on state machine definitions. The system measures the code coverage that each test script achieves when fed to the tested user interface of application program interface (API). Coverage metric is used as the fitness value.

Another patent has been issued to Whitten [6] at Sun Microsystems for a method for selecting a set of test cases which may be used to test software program products. The set of test cases is generated by the designer in the form of software that exercises as many of the code blocks in the product as possible. A GA is applied to determine which subset of test cases to use. This is done on the basis of the fitness value using a combination of time and coverage measurements. The aim is to determine the set of test cases that exercises a maximum number of code blocks in the minimum time.

Gounares and Sikchi [8] at Microsoft Corporation have received a patent for a system for adaptively solving sequential problems in a target software system utilizing modified GAs. Stimuli to the target system are presented as actions, and a sequence of actions is a GA chromosome. These chromosomes are applied to the target system one action at a time and the changes in properties of the target are measured. The fitness value is defined so that successive generations of chromosomes will converge upon the desired characteristics. For software testing these characteristics are defect discovery and code coverage. Evolving ever-shorter chromosomes that produce the same defect minimizes defects in a target software system, and the defect discovery rate is thereby maximized.

Michael and McGraw [44–46] have developed the so-called genetic algorithm data generation tool (GADGET) system that is fully automatic and supports all C/C++ constructs. The system is used to obtain condition/decision coverage. Michael et al. use triangle classification and an autopilot control program for a Boeing 737 as example problems. They also studied the performance of different GA variants and compared results with the random method—GAs gained a much higher coverage than the random method.

Pargas et al. [47] have experimented with genetic algorithm-based test data generation for statement and

branch coverage using a control-dependence graph to guide optimization. They tested six relatively small test programs and compared the results to the random method. Their approach clearly outperformed the random method for three of the six test programs, for the other three programs both methods find the optimal solution in the initial population. They suggest that the use of GA could be more beneficial for complex programs.

Bueno and Jino [48] have studied the possibility of using a GA to identify the potentially infeasible program paths. They propose that monitoring the progress of the GA search could identify an infeasible path. Their approach combines earlier works by other authors and introduces a new fitness function using control and data flow information to guide the search. They use the so-called ''path similarity metric'' as their fitness function. Results with their six small test programs were promising. The GA-based approach reached 100% success rate in unfeasible path identification in tenth of the amount of command executions and time needed by random search to reach 70% success rate.

2.3. Test data generation

Hunt [49] used a GA for testing cruise control system software. In his implementation a GA chromosome represents the input and corresponding expected output. The fitness value is assigned, if the measured output differs from the expected output. The greater the difference, the higher the fitness value. The expected output is derived from the original software specification. Hunt states that software is often developed by a third party, and the tester only has the software, which he treats as a black box and tests against the corresponding requirement specification. A GA chromosome must be able to represent all input values that the software can process, as well as the values that its single output can have. He claims that the chromosome must be able to represent both the valid and erroneous inputs. In his approach the GA is used as an aid for a human tester. The GA identifies failure scenarios, but it is up to the human tester to identify the faults that led to the failure.

Yang [10,50] has done a Ph.D. thesis on using genetic algorithms to derive test cases and test data from the formal Z specifications in order to test the functional behavior of the software. His aim was to show the conformance of the implementation to its specifications, i.e. the correctness of the implementation with respect to the set of test data with which it was exercised.

Minohara and Tohma [51] have used a GA for parameter estimation of a so-called ''hyper-geometric distribution software reliability growth model'', where the increase of the number of errors is observed as a function of time. Their GA chromosome represents a set of parameter values. The fitness value is evaluated by testing errors between the observed and the estimated test-and-debug data. They are trying to minimize the amount of errors. Their results suggest that the GA may be a more stable method to get the estimates.

Lin and Yeh [52] have also studied automatic test data generation by a GA for a chosen subpath. Their method uses a so-called ''normalized extended Hamming distance'' to guide the optimization process and to test the optimality of the candidate solutions. This fitness function, called SIMILARITY, defines how similar the traversed path is to the target path, is used to choose the surviving test cases. Optimality here means that the test case (i.e. a particular input) forces the program to follow the given path of program statements when executed. They claim that a GA is able to significantly reduce the time required for automatic path testing.

2.4. Testing program dynamics

Kasik and George [53] have used a GA for emulating software inputs in an unexpected, but not totally random way. The GA is used as a repeatable technique for generating user events that drive conventional automated test tools, so that the system can mimic different forms of novice user behavior. The system tries to represent how a novice user learns to use an application. The fitness value is given according to how much the actions performed are guided by the chromosome to resemble novice-like behavior. The novice behavior is described by a special reward system that was build based on observations.

Boden and Martino [54] used a GA to generate API tests. They concentrated on the operating system error treatment routines. The fitness function was a

weighted sum of various factors of a test response with an attempt to assess the sequences of operating system calls. Boden [7] has also received a US patent for an order-based GA-based automated testing of software application interface. The GA is used to search and detect symptoms of software errors by generating test sequences.

Wegener and co-workers [55–58] have studied the search of the execution time extremes of real-time software with a GA. They have compared their results to the random testing and static analysis. Their object software has mainly been some small examples or DaimlerChrysler embedded automotive electronics software. They think that the static analysis and evolutionary testing together can effectively find the lower and upper execution time limits. They claim that there is not much support for temporal testing, and often testers just use the methods that are designed to test the logical correctness. In their research the GA-based testing was much more effective than the random testing, and particularly effective when a problem has many variables and a large input domain. In their studies they measure execution times as processor cycles, so that interruptions etc. would not have an effect on results. A few times their GA found more extreme time than was previously known, the results were verified by analyzing the control flow graph. They also introduced the term ''evolutionary testing'', which means by their definition: ''the use of metaheuristic search methods for test case generation''.

Puschner and Nossal [59] have applied a GA for test data generation for testing WCET. Tests were executed in a simulation environment on a workstation and compared against the random testing, best effort data generation, and the static WCET analysis. The GA results compared well with the static WCET analysis, and clearly outperformed the random testing. They conclude that the GA is well suited for the WCET tests, and with large input spaces the GA-based method proved to be particularly favorable. See also our toy example in Section 1.8.

Ostrowski and Reynolds [60] present the implementation of the so-called cultural algorithms (CA) embedded with both the white and black box testing techniques. Cultural algorithms are GAs that has the so-called belief space that is used to pass the culture component, e.g. the acquired knowledge or accumulated experiences, from generation to generation. The idea is that the faults diagnosed by CA that does black box testing are passed to the CA that does the white box testing. The goal is automatic detection and isolation of program faults.

Pohlheim and coworkers [61,62] applied extensions of evolutionary algorithms (EA), called different strategies and competing subpopulations to automatic software testing. Several variants of GA are competing with each other and the best result is selected as the final solution of this technique. The object software was a DaimlerChrysler engine control software module, and the goal was to perform structure-oriented testing and temporal testing of real time software modules. The EA results were compared to the results obtained by the software developer with white box testing. The EA based automated test found always equal or even better execution times. It was observed that different EA strategies are successful with different software modules, each of the strategies were particularly successful at a specific point in time. By using competitive subpopulations with different EA strategies one can exclude unsuccessful ones.

The Ph.D. thesis of Gross [11] concentrates on the temporal B/WCET testing of real-time systems. He tries to measure ''evolutionary testability'' by studying if there is a relationship between the complexity of the test objects and the quality of the outcome produced by evolutionary testing. The example programs are short routines written in C++. Gross states that complexity as it is seen by the evolutionary algorithm is not much different from the way humans may experience it. This means that programs that were difficult for human analysis were also difficult for evolutionary testing.

The Ph.D. thesis by Tracey [12] deals with automatic test data generation for testing safety-critical systems. He uses simulated annealing and genetic algorithms, but also random search and hill climbing as the optimization methods. He defines the framework on how to use them for generating test data for temporal WCET testing, assertion based testing, and structural testing. It is observed that ''genetic algorithm-based approaches for structural test data generation have a number of weaknesses that restricts their application to real software industry''. On the other hand, GAs seems to be, on average, the most effective and efficient of the techniques he implemented in his work.

2.5. Black box testing

Bingul et al. [63] apply a GA to test the war simulation software THUNDER with the black box method. They applied multiobjective optimization with the Pareto method, and define three different ways to assign fitness values. They try to optimize software behavior, war strategies, and the running time. They claim that the GA was able to provide optimal or near optimal solutions.

We have worked with genetic algorithm-based software testing since 1995. The first application was the testing of a large embedded software [64–67,13]. The object software was received from our industrial partner, and the optimization goal was to find the extreme timing conditions, in other words the input combinations that caused the software response times increase maximally long. The tested input domain consisted of CAN and LON communication, and digital measurement inputs. The testing was done using the pure black box testing scheme: not any code was traced, only the communication that the software did with outside world and the timing of those communications. The only fitness value the GA received was the response time, the time which software needed to reply to the input values generated by the GA. The results showed that the GA is able to learn input domain values that lead to the longer response times than just by using the random testing. From the results we were able to draw some conclusions of what effect the messages, and their timings have on each other.

Next, we applied the same principles for testing image processing and vision softwares with test images and surfaces generated by genetic algorithms [68,69]. The results of this work seems to confirm that the GA is capable of generating test images and surfaces that reveal some weaknesses in the image processing or measurement software. To check the approach we also did some error seeding. The seeded errors were effectively found by using the optimized test images. The test surface generation also revealed the type of surfaces that have highest number of errors.

Our latest work has concentrated on applying co-evolution in order to develop the software parameters simultaneously with the testing [70]. In order to do that, the object software must have some parameters that we can change. The results seem to show that when we can tune software parameters simultaneously with the testing: a co-evolutionary GA can generate the worst test case to the population of software versions with the current parameter setup, while also optimizing the parameter setups, so that the current software population handles the worst test case as well as possible [13].

2.6. Software quality

Hochman et al. [71,72] applied a GA to optimize neural network architectural, learning, and training parameters. They call the method ‘‘evolutionary neural networks’’, and use them to detect fault-prone and not-fault-prone software modules. They compare their method against the discriminant analysis to discover software reliability problems. The statistical analysis of their results seems to confirm that their approach is superior.

Mansour and coworkers [73,74] applied a GA to the optimal regression-testing problem. They try to determine the minimum number of test cases for revalidating modified software in the maintenance phase. They compare the GA-based method with the branch and bound (B & B) and simulated annealing (SA) based methods. The B & B method was the fastest, when testing small modules, but as the size increases, and the number of test cases grows, the GA-based method becomes fastest. They conclude that in contrast to analysis-based optimization methods, the complexity of the GA and SA does not grow exponentially with module sizes. Results show that the GA and SA find an optimal or nearly optimal number of regression testing tests in a reasonable time.

Baisch and Liedge [75,76] used GAs for tailoring a fuzzy rule base of an expert system used for software quality prediction. Their object software is from a large real-time telecommunication system. The GA is used to classify the software modules into two classes: (1) few faults (<5), (2) many faults (>20). The authors discovered that additional factors, like fault history, change history, and size should be utilized. There were also several faults that cannot be predicted by the system. They claim that the proposed system helps to decrease faults by up to 50% after changes in the modules.

Evett et al. [88] used genetic programming (GP) approach for software quality prediction. Their system

predicts the relative quality of each module, instead of a classic classification into fault-prone and not-fault-prone modules. Their GP used the size of code, degree of reuse, and faults in the previous releases to predict the number of expected faults in each module. Their target was two industrial softwares, a large military communication system, written in ADA, and a large telecommunication system, written in a Pascal like language. Both systems contain approximately 200 modules, from which they use two-thirds as training data and the rest for validating the predictive accuracy of the best model developed. Their conclusion was that GP is able to generate software quality models based on data collected earlier in the development phase.

Burges and Lefley [77] applied GP to the estimation of a software project effort. They use data collected from existing software projects, and generate estimation models for these with GP. The estimation is done by using data available from the specification stage. They compare their GP based method against the statistical and neural network based methods. It is concluded that while the GP and artificial neural networks (ANN) are able to provide better accuracy, they require more effort for set up and training.

Aguilar-Ruiz et al. [78] used evolutionary algorithms to estimate software development projects. They use a software project simulator that generates a database from the software project. EA is then used to produce a set of management rules. The aim is that these rules will help the project manager to keep the project within the budget, and to reach the quality and duration targets. The EA generated rules are generated against rules generated by a commonly used C4.5 tool that uses a recursive algorithm that optimizes rules from decision trees. The practical results seem to indicate that the approach using the EA finds better solutions.

Jones et al. have produced several papers on using GAs in testing [50,55,56,79–84]. In one article [84] they call this new field of software engineering research ''search-based software engineering''. They argue that software engineering is ideal for the application of metaheuristic search techniques. They also note that the search-based technique must outperform the random technique in order to be qualified as worthy of even being considered a successful application. The random method therefore provides the lowest benchmark. If the metaheuristic method does not outperform the random method, it is likely that the reason is that it is poorly implemented. They also expect to see a dramatic growth in the field of search-based software engineering within the next few years. They list the likely application areas and the developments that the growing research capacity will provide.

For further references of evolutionary optimization methods in software engineering see bibliography [4].

3. Discussion

The researchers in the field of evolutionary software engineering usually tend to study the testing of their private object software, that is usually obtained from industrial partners, therefore comparing results is problematic if not impossible. The only commonly used benchmark problem seems to be the triangle classification problem. The problem is to classify whether three given numbers a, b, and c (length of triangle edges) form a triangle and of which type; sharp, straight or obtuse-angled. This problem is used with the white box testing strategy, the path or condition coverage is usually used as the corresponding optimization target.

The genetic algorithm does not find any solitary bug from the software with any higher probability than the random search (see Fig. 7). However, if the bug is

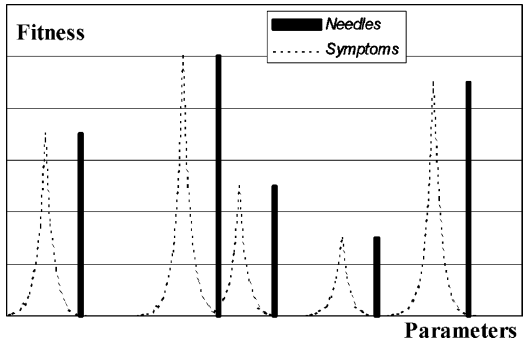

Fig. 7. *Needles*: needles in a haystack type of problem. Software bugs do not have any symptoms in the problem neighborhood. The GA cannot adapt to the environment. *Symptoms*: a fitness landscape that does indicate symptoms in the neighborhood of the problematic situation. When this occurs GA can adapt to the environment and gain advantage over random methods.

composed of a combination of input parameters, or a chain of events that are caused by them, then the GA gains advantage. In practice this means that a GA tester generates several parameter combinations that cause minor bugs and further constructs an input sequence that will have a higher amount of instances that cause more minor bugs than the pure random testing. When testing nondeterministic time critical software with a temporal fitness function, the GA recognizes the inputs that cause delays, it starts to favor those parameters, and eventually generates input that causes maximum response delay. Note that the maximum or minimum execution time found cannot be guaranteed to be the global extreme. There are methods like static analysis that can be used to validate the extreme cases, but with a large software it comes virtually impossible.

The GA-based testing do not need separate boundary or zero testing, e.g. if a division by zero exist in the software, the GA recognizes the symptoms when near zero and soon reveals that error situation. This however depends on the software metric that is used, if we want to check boundaries and division by zero, we must define software metric that reveals the symptoms also for the neighboring values.

A GA has also been applied to mutation testing [80], where it effectively killed the mutants, the boundaries of which were changed.

The advantages of the GA-based approach over random testing include:

(1) needs less human analysis, the GA pre-analyses the software according to the fitness function,
(2) automatically tests combinations of suspicious parameters, and
(3) may find a combination of input that leads to a more severe fault behavior.

One of the most beneficial features of the GA is that it can be easily applied to a variety of different problems. This applies to the software engineering and testing, too. The automated testing algorithm can be easily adapted to test other subroutines: the optimization algorithm itself does not need to be modified at all. Only the fitness function and the part of the program that inputs test data to the target software must be prepared so that it is compatible with the interface of the new target. The chromosomes of the GA are converted into corresponding parameters of the tested subroutine. Hence the cost to automate testing of every new routine in a software project with a GA-based optimization approach should be lower than the estimations represented in [85–87], and it should be at least partly possible to make it automatically. The GA-based approach may be best suited for the timings and stress testing that are also considered the testing types best suited for automation in general.

One possible drawback of using the GAs with black box testing method is that it by no means guarantee code coverage. The GAs heavily favor those inputs that cause high fitness value with the given software metric, but may not cover much of the total search space.

3.1. Fitness landscape

The obvious drawback of the evolutionary approach is that solitary errors—needles in a hay stack—cannot be detected by any greater efficiency than using, e.g. random search. In this respect all black-box type testing methods are equivalent. Only white-box type methods have the possibility to reveal such errors if they are related to some features of the program structure. In practise most errors are not solitary but appear in some context having some finite extend. In other words the fitness landscape has some correlation between neighboring points. The longer the correlation length the more efficient is search based testing when compared to random testing. Especially this applies to testing of program execution time extremes and other similar stress testing. Unfortunately not much is known of the properties of fitness landscapes neither in testing software nor in optimization in general.

4. Conclusions and future

All researchers that have used genetic algorithm-based optimization methods in the software testing area have reported ‘‘good’’, ‘‘excellent’’, ‘‘positive’’ or at least ‘‘encouraging’’ results. The lack of negative results might be partly due to the fact that researchers do not want to release negative results. However, the sheer expanding amount of research activity in the

field [4] seems to suggest that this method is a serious contender. The fact that many researchers that are now active in this field have earlier used more traditional software testing methods also strengthen this view.

GAs are well applicable to problems that are discrete or have no exact mathematical expressions or models as is the case of software testing. Harman and Jones [84] claimed that software engineering is ideal for evolutionary based optimization, none of our findings contradicts this claim.

The effectiveness of GAs tend to depend on implementation details, how the problem is encoded etc. If GA does not seem to outpower the random method, it is usually implemented poorly or in the wrong way, or the GA operators: crossover, mutation, selection schemes have been improperly selected or have pathological values. However, GAs are robust, small changes in GA parameter values do not usually affect much the optimization efficiency or result. When properly implemented GA is highly applicable for software testing: testing coverage, timings, parameter values, or finding calculation tolerances, bottlenecks, problematic input combinations, and sequences. If program parameters are changeable on the fly, co-evolutionary GA can tune the tested software during the testing.

We propose that GA-based automatic testing tools can be used to automatically generate test data for module testing. Although a random generator can just as simply generate the test data, a GA-based testing more easily reveals problematic parameters and starts to construct more erroneous situations using only a fraction of the test cases that the pure random search method generates.

We expect that the studies involving evolutionary optimization based software engineering will increase in the following years and many of the questions still unanswered will be studied.

Acknowledgements

Our thanks to Lilian Rautiainen for her comments concerning English writing and Sakari Kauvosaari for maintaining Vaasa bibliography database. Technical Research Institute of the University of Vaasa is acknowledged of financing this research work. Referees are acknowledged for their valuable comments.

References

[1] J.T. Alander, Indexed bibliography of genetic algorithms basics, reviews, and tutorials, Report 94-1-BASICS, Department of Information Technology and Production Economics, University of Vaasa (ftp://uwasa.fi/cs/report94-1/gaBASICS-bib.ps.Z), 1995.

[2] J.T. Alander, Indexed bibliography of genetic programming, Report 94-1-GP, Department of Information Technology and Production Economics, University of Vaasa (ftp://uwasa.fi/cs/report94-1/gaGPbib.ps.Z), 1995.

[3] J.T. Alander, Indexed bibliography of genetic algorithms in electronics and VLSI design and testing, Report 94-1-VLSI, Department of Information Technology and Production Economics, University of Vaasa (ftp://uwasa.fi/cs/report94-1/gaVLSIbib.ps.Z), 1995.

[4] J.T. Alander, Indexed bibliography of genetic algorithms in computer science, Report 94-1-CS, Department of Information Technology and Production Economics, University of Vaasa (ftp://uwasa.fi/cs/report94-1/gaCSbib.ps.Z), 1995.

[5] R.W. Warfield, Automatic software testing tool, U.S. patent no. 5,754,760, issued May 19, 1998.

[6] T.G. Whitten, Method and computer program product for generating a computer program product test that includes an optimized set of computer program product test cases, and method for selecting same, U.S. patent no. 5,805,795, issued September 8, 1998.

[7] E.B. Boden, Automated testing of software application interfaces, object methods and commands, U.S. patent no. 5,708,774, issued January 13, 1998.

[8] A. Gounares, P. Sikchi, Adaptive problem solving method and apparatus utilizing evolutionary computation techniques, U.S. patent no. 6,282,527, issued August 28, 2001.

[9] H.-H. Sthamer, The automatic generation of software test data using genetic algorithms, Ph.D. thesis, Department of Electronics and Information Technology, University of Glamorgan, 1996.

[10] X. Yang, Automatic testing from Z specifications, Ph.D. thesis, University of Glamorgan, 1998.

[11] H.-G. Gross, Measuring evolutionary testability of real-time software, Ph.D. thesis, University of Glamorgan, 2000.

[12] N. Tracey, A search-based automated test-data generation framework for safety-critical software, Ph.D. thesis, University of York, 2000.

[13] T. Mantere, Automatic software testing by genetic algorithms, Ph.D. thesis, Acta Wasaensia 112, Department of Computer Science, University of Vaasa, 2003.

[14] J.H. Holland, Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, 1975.

[15] D.E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, 1989.

[16] C. Darwin, The Origin of Species: By Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life, Oxford University Press, London, UK, 1859 (http://www.literature.org/authors/darwin-charles/the-origin-of-species/).

[17] J.R. Koza, Genetic Programming, The MIT Press, Cambridge, MA, 1992.

[18] T.C. Jones, Measuring programming quality and productivity, IBM Syst. J. 17 (1978) 39–63.
[19] B.W. Boehm, Some Steps Towards Formal and Automated Aids to Software Requirements Analysis and Design, IFIB74, North-Holland Publishing Company, Amsterdam, The Netherlands, 1974.
[20] E. Kit, Software Testing in the Real World—Improving the Process, Addison-Wesley, 1995.
[21] IEEE/ANSI, IEEE Standard Glossary of Software Engineering Terminology IEEE Std 620.12-1990, IEEE/ANSI, 1990.
[22] IEEE/ANSI, IEEE Standard for Software Test Documentation, IEEE Std 829-1983, IEEE/ANSI, 1983.
[23] Lonsdale Systems, Software testing glossary (members.iinet.net.au/lonsdale/seng/se20.htm), 2002.
[24] R.S. Pressman, Associates Inc., An abbreviated software engineering glossary (http://www.rspa.com/spi/glossary.html), 2003.
[25] H.-G. Beyer, E. Brucherseifer, W. Jakob, H. Pohlheim, B. Sendhoff, T. Binh, Evolutionary algorithms—terms and definitions (ls11-http://www.cs.uni-dortmund.de/people/beyer/EA-glossary/def-engl-html.html), 2002.
[26] E. Miller, Program testing—an overview for managers (IEEE software testing tutorial), 1980.
[27] R.L. Baber, Software Reflected, North-Holland Publishing Company, Amsterdam, The Netherlands, 1982.
[28] F. Liguori (Ed.), Automatic Test Equipment: Hardware, Software, and Management, IEEE Press, New York, USA, 1972.
[29] W. Hetzel, The Complete Guide to Software Testing, second ed. John Wiley and Sons Inc., New York, USA, 1988.
[30] N.D. Singpurwalla, S.P. Wilson, Statistical Methods in Software Engineering—Reliability and Risk, Springer-Verlag, New York, USA, 1999.
[31] J.C. Browne, M. Shaw, Toward a scientific basis for software evaluation, in: A. Perlis, F. Sayward, M. Shaw (Eds.), Software Metrics: An Analysis and Evaluation, The MIT Press, Cambridge, Massachusetts, USA, 1981.
[32] M. Denicoff, R. Grafton, Software metrics: a research initiative, in: A. Perlis, F. Sayward, M. Shaw (Eds.), Software Metrics: An Analysis and Evaluation, The MIT Press, Cambridge, Massachusetts, USA, 1981.
[33] S. Xanthakis, C. Ellis, C. Skourlas, A.L. Gall, S. Katsikas, K. Karapoulios, Application of genetic algorithms to software testing, in: Proceedings of the 5th International Conference on Software Engineering, Toulouse, France, 1992, pp. 625–636.
[34] A.C. Schultz, J.J. Grefenstette, K.A.D. Jong, Adaptive testing of controllers for autonomous vehicles, in: Proceedings of the 1992 Symposium on Autonomous Underwater Vehicle Technology (AUV'92), IEEE, New York, Washington, DC, 1992, pp. 158–164.
[35] A.C. Schultz, J.J. Grefenstette, K.A.D. Jong, Test and evaluation by genetic algorithms, IEEE Expert 8 (5) (1993) 9–14.
[36] A.C. Schultz, J.J. Grefenstette, K.A.D. Jong, G3.5 learning to break things: adaptive testing of intelligent controllers, 1995, pp. G3.5:1–10.
[37] M. Pei, E.D. Goodman, Z. Gao, K. Zhong, Automated software test data generation using a genetic algorithm, Technical report 6/2/94, Beijing University of Aeronautics and Astronautics, 1994.
[38] B. Korel, Automated software test data generation, IEEE Trans. Software Eng. 16 (8) (1980) 870–879.
[39] A.L. Watkins, The automatic-generation of test data using genetic algorithms, in: I.M. Marshall, W.B. Samson, D.G. Edgar-Nevill (Eds.), Proceedings of the 4th Software Quality Conference, vol. 2, University of Abertay Dundee, Scotland, Dundee, UK, 1995, pp. 300–309.
[40] M. Roper, I. Maclean, A. Brooks, J. Miller, M. Wood, Genetic algorithms and the automatic generation of test data, Research report RR/95/195 [EFoCS-19-95], Department of Computer Science, University of Strathclyde, 1995.
[41] M. Roper, Software testing—searching for the missing link, Inform. Software Technol. 41 (14) (1999) 991–994.
[42] J.E. Smith, T.C. Fogarty, Evolving software test data—GA's learn self expression, in: Proceedings of the Evolutionary Computing on AISB Workshop, Springer-Verlag, Berlin (Germany), Brighton (UK), 1996, pp. 227–235.
[43] J.E. Smith, M. Bartley, T.C. Fogarty, Microprocessor design verification by two-phase evolution of variable length tests, in: Proceedings of the IEEE International Conference on Evolutionary Computation, IEEE Piscataway, NJ, Indianapolis, IN, 1997, pp. 453–458.
[44] C.C. Michael, G.E. McGraw, M.A. Schatz, C.C. Walton, Genetic algorithms for dynamic test data generation, in: Proceedings of the 12th IEEE International Conference Automated Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, Incline Village, NV, USA, 1997, pp. 307–308.
[45] G.E. McGraw, C.C. Michael, M.A. Schatz, Generating software test data by evolution, Technical report RSTR-018-97-01, RST Corporation, 1998.
[46] C.C. Michael, G.E. McGraw, M.A. Schatz, Generating software test data by evolution, IEEE Trans. Software Eng. 27 (12) (2001) 1085–1110.
[47] R.P. Pargas, M.J. Harrold, R.R. Peck, Test-data generation using genetic algorithms, J. Software Test. Verif. Reliab. 9 (4) (1999) 263–282.
[48] P.M.S. Bueno, M. Jino, Identification of potentially infeasible program paths by monitoring the search for test data, in: Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering, ASE 2000, IEEE Piscataway, NJ, Grenoble, France, 2000, pp. 209–218.
[49] J. Hunt, Testing control software using a genetic algorithm, Eng. Appl. Artif. Intell. 8 (6) (1995) 671–680.
[50] B.F. Jones, H.-H. Sthamer, X. Yang, D.E. Eyres, The automatic generation of software test data sets using adaptive search techniques, in: Proceedings of the Software Quality Management 3, Computational Mechanics Publications Ltd., Southhampton, UK, Seville, Spain, 1995, pp. 435–444.
[51] T. Minohara, Y. Tohma, Parameter estimation of hyper-geometric distribution software reliability growth model by genetic algorithms, in: Proceedings of the Sixth International Symposium on Software Reliability Engineering, IEEE Piscataway, NJ, Toulouse, France, 1995, pp. 324–329.

[52] J.-C. Lin, P.-L. Yeh, Automatic test data generation for path testing using GAs, Inform. Sci. 131 (1–4) (2001) 47–64.
[53] D.J. Kasik, H.G. George, Toward automatic generation of novice user test scripts, in: Proceedings of the 1996 Conference on Human Factors in Computing Systems, CHI 96, ACM, New York, NY, Vancouver, BC, Canada, 1996, pp. 244–251.
[54] E.B. Boden, G.F. Martino, Testing software using order-based genetic algorithms, in: J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo (Eds.), Proceedings of the GP-96 Conference, MIT Press, Cambridge, MA, Stanford, CA, 1996, pp. 461–466.
[55] J. Wegener, M. Grochtmann, B. Jones, Testing temporal correctness of realtime systems by means of genetic algorithms, in: Proceedings of the 10th International Software Quality Week (QW '97), San Francisco, CA, 1997.
[56] B.F. Jones, J. Wegener, Measurement of extreme execution times for software, in: IEE Colloquium on Real-Time Systems, vol. 306, IEE, 1998, pp. 4/1–4/5.
[57] J. Wegener, M. Grochtmann, Verifying timing constraints of real-time systems, Real-Time Syst. 15 (3) (1998) 275–298.
[58] J. Wegener, A. Baresel, H. Sthamer, Evolutionary test enviroment for automatic structural testing, Inform. Software Technol. 43 (2001) 841–854.
[59] P. Puschner, R. Nossal, Testing the results of static worst-case execution-time analysis, in: Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS '98), IEEE Piscataway, NJ, Madrid, Spain, 1998, pp. 134–143.
[60] D.A. Ostrowski, R.G. Reynolds, Knowledge-based software testing agent using evolutionary learning with cultural algorithms, in: Proceedings of the 1999 Congress on Evolutionary Computation, CEC '99, IEEE Piscataway, NJ, Washington, DC, 1999, pp. 1657–1663.
[61] J. Wegener, H. Sthamer, H. Pohlheim, Testing the temporal behaviour of realtime tasks using extended evolutionary algorithms, in: Proceedings of the 7th European Conference on Software Testing, Analysis and Review (EuroSTAR '1999), Barcelona, Spain, 1999.
[62] H. Pohlheim, Competition and cooperation in extended evolutionary algorithms, in: L. Spector (Ed.), Gecco 2001—Late Breaking Papers, Morgan Kauffman, San Francisco, CA, 2001.
[63] Z. Bingul, A.S. Sekmen, S. Palaniappan, S. Zein-Sabatto, Genetic algorithms applied to real time multiobjective optimization problems, in: Proceedings of the IEEE Southeastcon, IEEE Piscataway, NJ, Nashville, TN, 2000, pp. 95–103.
[64] J.T. Alander, T. Mantere, P. Turunen, Genetic algorithm based software testing, in: G.D. Smith, N.C. Steele, R.F. Albrecht (Eds.), Artificial Neural Nets and Genetic Algorithms, Proceedings of International Conference (ICANNGA'97), Springer-Verlag, Wien, Norwich, UK, 1998, pp. 325–328.
[65] J.T. Alander, T. Mantere, G. Moghadampour, Testing software response times using a genetic algorithm, in: J.T. Alander (Ed.), Proceedings of the Third Nordic Workshop on Genetic Algorithms and their Applications (3NWGA), Finnish Artificial Intelligence Society (FAIS), Helsinki, Finland, 1997, pp. 293–298, (ftp://uwasa.fi/cs/3NWGA/*.ps.Z).
[66] J.T. Alander, T. Mantere, G. Moghadampour, J. Matila, Searching protection relay response time extremes using genetic algorithm—software quality by optimization, Electr. Power Syst. Res. 46 (1998) 229–233.
[67] J.T. Alander, T. Mantere, Automatic software testing by genetic algorithm optimization, a case study, in: C. Ryan (Ed.), Proceedings of the First International Workshop on Soft Computing Applied to Software Engineering, Limerick University Press, Limerick, Ireland, 1999, pp. 1–9.
[68] T. Mantere, J.T. Alander, Automatic test image generation by genetic algorithms for testing halftoning methods, in: D.P. Casasent (Ed.), Intelligent Systems and Advanced Manufacturing: Intelligent Robots and Computer Vision XIX: Algorithms, Techniques, and Active Vision, vol. SPIE-4197, The International Society for Optical Engineering, Bellingham, WA, Boston, MA, 2000, pp. 297–308.
[69] T. Mantere, J.T. Alander, Testing halftoning methods by images generated by genetic algorithms, Arpakannus 1 (2001) 39–44.
[70] T. Mantere, J.T. Alander, Developing and testing structural light vision software by co-evolutionary genetic algorithm, in: Proceedings of the Second ASERC Workshop on Quantitative and Soft Computing based Software Engineering, Alberta Software Engineering Research Consortium (ASERC), Banff, Canada, 2002, pp. 31–37.
[71] R. Hochman, T.M. Khoshgoftaar, E.B. Allen, J.P. Hudepohl, Using the genetic algorithm to build optimal neural networks for faultprone module detection, in: Proceedings of the Seventh International Symposium on Software Reliability Engineering, IEEE Computer Society Press, Los Alamitos, CA, White Plains, NY, USA, 1996, pp. 152–162.
[72] R. Hochman, T.M. Khoshgoftaar, E.B. Allen, J.P. Hudepohl, Evolutionary neural networks: a robust approach to software reliability problems, in: Proceedings of the Eighth International Symposium on Software Reliability Engineering, IEEE Computer Society Press, Los Alamitos, CA, Albuquerque, NM, 1997, pp. 13–26.
[73] N. Mansour, K. El-Fakih, Natural optimization algorithms for optimal regression testing, in: Proceedings of the 1997 21st Annual International Conputer Software & Applications Conference, IEEE, Los Alamitos, CA, Washington, DC, 1997, pp. 511–514.
[74] G. Baradhi, M. Nashat, Comparative study of five regression testing algorithms, in: Proceedings of the 1997 Australian Software Engineering Conference, IEEE Computer Society Press, Los Alamitos, CA, Sydney, NSW, Australia, 1997, pp. 174–182.
[75] E. Baisch, T. Liedtke, Comparison of conventional approaches and soft-computing approaches for software quality prediction, in: Proceedings of the 1997 IEEE International Conference on Systems, Man, and Cybernetics, IEEE, Piscataway, NJ, Orlando, FL, 1997, pp. 1045–1049.
[76] E. Baisch, T. Liedtke, Automated knowledge acquisition and application for software development projects, in: 13th IEEE Conference on Automated Software Engineering, IEEE Piscataway, NJ, Honolulu, HI, 1998, pp. 13–16.
[77] C.J. Burges, M. Lefley, Can genetic programming improve software effort estimation? A comparative evaluation Inform. Software Technol. 43 (14) (2001) 863–873.

[78] J.S. Aguilar-Ruiz, I. Ramos, J.C. Riquilme, M. Toro, An evolutionary approach to estimating software development projects, Inform. Software Technol. 43 (14) (2001) 875–882.
[79] H.-G. Gross, B.F. Jones, D.E. Eyres, Structural performance measure of evolutionary testing applied to worst-case timing of real-time systems, IEE Proceedings: Software 147 (2) (2000) 25–30.
[80] B.F. Jones, H.-H. Sthamer, D.E. Eyres, Automatic structural testing using genetic algorithms, Software Eng. J. 11 (5) (1996) 299–306.
[81] B.F. Jones, D.E. Eyres, H.-H. Sthamer, A strategy for using genetic algorithms to automate branch and fault-based testing, Comput. J. 41 (2) (1998) 98–107.
[82] H.-G. Gross, B.F. Jones, D.E. Eyres, Evolutionary algorithms for the verification of execution time bounds for real-time software, in: IEE Colloquium on Applicable Modelling, Verification and Analysis Techniques for Real-Time Systems (ref. no. 1999/006), IEEE Piscataway, NJ, London, UK, 1999, pp. 8/1–8/8.
[83] H.-H. Sthamer, B.F. Jones, D.E. Eyres, Generating test data for ADA generic procedures using genetic algorithms, in: Proceedings of ACEDC 1994, University of Plymouth, UK, 1994, pp. 134–140.
[84] M. Harman, B.F. Jones, Search-based software engineering, Inform. Software Technol. 43 (2001) 833–839.
[85] C. Kaner, Improving the maintainability of automated test suites, in: Proceedings of the Tenth International Quality Week on Software Research, San Francisco, CA, 1997.
[86] B. Marick, Classic testing mistakes, in: Proceedings of STAR 97, Sixth International Conference on Software Testing, Analysis, and Review, San Jose, CA, 1997.
[87] B. Pettischord, Success with test automation, in: Proceedings of the Ninth International Quality Week on Software Research, San Francisco, CA, 1996.
[88] M. Evett, T. Khoshgoftar, P. Chien, E. Allen, GP-based software quality prediction, in: Proceedings of the Third Annual Conference Genetic Programming, Morgan Kauffman, Madison, WJ, pp. 60–65.