How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution

Konstantinos Adamopoulos, Mark Harman, and Robert M. Hierons

Department of Information Systems and Computing,
Brunel University,
Uxbridge, Middlesex, UB8 3PH, UK
konstantinos.adamopoulos@brunel.ac.uk

Abstract. The use of Genetic Algorithms in evolution of mutants and test cases offers new possibilities in addressing some of the main problems of mutation testing. Most specifically the problem of equivalent mutant detection, and the problem of the large number of mutants produced. In this paper we describe the above problems in detail and introduce a new methodology based on co-evolutionary search techniques using Genetic Algorithms in order to address them effectively. Co-evolution allows the parallel evolution of mutants and test cases. We discuss the advantages of this approach over other existing mutation testing techniques, showing details of some initial experimental results carried out.

1 Introduction

Software testing is a vital yet expensive part of the software development process. Studies suggest that it often consumes in the order of fifty percent of the total development budget [1,2]. Thus, approaches that automate or semi-automate sections of software testing may significantly improve the efficiency and quality of the software development process.

1.1 Overview of Mutation Testing

Mutation testing is a software testing technique originally proposed by Hamlet [3]. Mutation testing is based upon seeding the implementation (original program) with a fault (*mutating it*), by applying a *mutation operator*, and determining whether testing identifies this fault. A mutated program is called a *mutant* and if a test case distinguishes between the mutant and the original program it is said to *kill* the mutant. Given a set of test cases, if no test case can distinguish between the mutant and the original program then the mutant is still *live*.

Mutation testing may be used to judge the effectiveness of a test set: the test set should kill all the mutants. Similarly, test generation may be based on mutation testing: tests are generated to kill the mutants. Interestingly, many test criteria may be represented using mutation testing by simply choosing appropriate mutation operators. Test sets are measured according to *mutation score*, or *adequacy score*, which is defined as the total number of killed mutants over the number of non-equivalent mutants, where a mutant is said to be *equivalent* if there does not exist a test case which can distinguish the output of the mutant from the output of the original program. Mutation score takes real values between 0.0 and 1.0, where 1.0 is the best score possible, meaning that this particular test set can kill all the non-equivalent mutants. Such a test set is said to be 100% mutation adequate.

Due to the fact that mutation testing is of high computational cost, even in the case of small and rather simple programs, several techniques were developed to reduce considerably the computational cost of effectiveness. Among these techniques are selective mutation [4], mutant sampling [5], weak mutation [6,7], schema-based mutation [8], separate compilation [9], and search. In this paper we examine some weak points of the 'do fewer' techniques, such as selective mutation and mutant sampling, and we introduce a new, search based, methodology based on Genetic Algorithms (GAs).

Selective mutation identifies the critical mutation operators that provide almost the same testing coverage as non-selective mutation. A number of mutants are rejected, because low performance mutation operators are rejected. As a result, the number of mutants is considerably decreased thereby reducing computational cost.

In a similar manner, mutant sampling is another approach to reduce the number of mutants under consideration. The main concept of this approach is the random selection of a subset of mutants; either by using samples of some *a priori* fixed size, or until sufficient evidence has been collected indicating that a statistically appropriate sample size has been reached [4,5].

The main assumption of the present work is that the 'do fewer' approaches (selective mutation and mutant sampling) have some weak points: Selective mutation does not take into consideration the fact that there are different sets of critical mutation operators for different classes of programs. On the other hand, mutant sampling is an approach that works satisfactorily with small randomly selected subsets of mutants but the testing coverage achieved is reduced [10].

While mutation testing is a powerful and general technique, a number of associated problems have limited its practical impact.

1. One of these problems is that the standard set of mutation operators may lead to a vast number of mutants, not always appropriate ones. Typically there is a large number of mutants for even small software units. Because of this, mutation testing is considered to be a computationally expensive method; all mutants must be tested against all test cases, leading to an increasing demand for computational resources. However, it has been noted that this problem can be reduced by an appropriate choice of mutation operators, using selective mutation [4].
2. Another important problem with mutation testing is that a mutant, while syntactically different from the initial program, may have the same behaviour

as the program. Clearly such *equivalent* mutants can never be killed and thus it is important, but often difficult, to identify them. Some approaches to detecting equivalent mutants and reducing the number of equivalent mutants produced have been introduced [11,12,13,14,15]. Manual equivalent mutant detection is tedious and new methodologies to automate this process should be introduced.

The contributions of this paper are to:

– Address the problem of selecting the effective mutants and test cases by searching for them.
– Reduce the large number of mutants produced and the associated computational cost, using natural selection, rather than artificial selection.
– Show how the fitness function can be designed to avoid generation of equivalent mutants.
– Explore how co-evolution can be used to automatically generate and evolve both mutants and test cases.

The first author to consider the application of genetic algorithms to mutation testing was Bottaci [16,17]. Many other authors have also suggested search for test data generation for other forms of testing [18,19,20,21,22,23,24,25,26,27,28]. However, this is the first paper to introduce the idea of co-evolution for mutation testing. A recent survey of work on evolutionary test data generation is provided by McMinn [29].

The problem of selection is a special case of the Feature Subset Selection Problem [30], where from an existing set of features there must be selected a subset of features that can achieve the same quality of properties. In our case to select the subsets of mutants and the corresponding subsets of test cases that provide the highest testing coverage possible. That is, as close as possible to 100% mutation adequacy.

In the present work we will present a three step methodology. The first step is to utilize a GA for the evolution of the mutants of a given program. The second step, similarly, is to utilize a GA for the evolution of the test cases of a given program. And finally, the third step which refers to the use of GAs for the co-evolution of both a population of mutants and a set of test cases for a given program.

The rest of the paper is organized as follows: Section 2 describes the proposed methodology. Section 3 presents some results of the experiments carried out. Section 4 outlines some of the outstanding issues for future consideration. Finally section 5 presents the conclusions.

2 Methods

In this section the three aspects of our methodology will be presented, namely the evolution of subsets of mutants against a fixed set of test cases, the evolution of subsets of test cases against a fixed set of mutants, and the co-evolution of both populations by evolving in parallel subsets of mutants and subsets of test cases.

2.1 Evolution of Subsets of Mutants Against a Fixed Set of Test Cases

In order to evolve subsets of mutants that are non-equivalent and at the same time difficult to kill first we generate a pool of mutants of the original program under test. From this pool our candidate mutants will be drawn. This is done by using a simulation of a mutation testing tool such as Mothra [31].

Each member of the constructed pool of mutants has been validated against a fixed set of test cases. This validation procedure associates each mutant with a corresponding score that is related to the performance of the mutant against a given fixed set of test cases. The higher the score, the more difficult it is for a mutant to be killed. In this way the score is a measure of the ability of the mutant to avoid being killed by the test cases. This validation procedure has been simulated. The simulation generates random real numbers between 0.0 and 1.0; each number represents a score associated with a mutant.

The main idea is to construct subsets of these mutants, each subset to be considered as an individual of the genetic algorithm. Therefore the initial population of the genetic algorithm consists of subsets of mutants that are randomly selected from the pool of mutants. The simulation does this by grouping a number of scores together; each group of scores correspond to a subset of mutants.

Evolution of this population with a GA will lead to subsets of mutants that are combined to give the maximum total score. This is done by selecting the fitness function of an individual to be the sum of the scores of the mutants divided by the length of the individual. That is, if S_i is the score of a mutant i and the individual consists of L mutants the fitness Mf of this individual is given by:

$$Mf = \begin{cases} \frac{\sum_{i=1}^{L} S_i}{L} & \text{if } \forall i.S_i \neq 1. \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

It is obvious that if there exists i such that $S_i = 1$, then there is a mutant killed by no test cases. This may be equivalent, that's why in this case the fitness function of this particular set of mutants is penalized by assigning to it the value 0. Also, the normalization of the fitness values, in order to get real values between 0.0 and 1.0, makes the fitness of individuals with different length comparable.

The fitness function, as described here, is relatively easy to optimise (even without a search technique). However, the evolution of subsets of mutants against a fixed set of test cases is only a preliminary milestone in the road towards the ultimate goal of achieving full co-evolution of both test set and mutant set. See Section 2.3.

We use uniform crossover. The mutation operator exchanges a single mutant of an individual with another mutant randomly selected from the pool of mutants. The selection operator is the roulette-wheel selection that ensures that the probability of selection of an individual is proportional to its fitness. Elitism was also introduced in the GA, that means a user defined number of highly fit individuals are automatically passed unchanged to the next generation.

By applying the GA we achieve to work with the whole set of mutation operators; there is no need to decrease the number of mutation operators, as in selective mutation. At the same time the GA achieves tailored selective mutation by genetically selecting sets of mutants with higher fitness values. That is, a higher chance of avoiding being killed. Additionally possible equivalent mutants are detected and eliminated.

2.2 Evolution of Subsets of Test Cases Against a Fixed Set of Mutants

We can apply the same methodology to the evolution of subsets of test cases as we applied to evolution of mutants. First, we randomly generate a pool of test cases from which our candidate test sets will be drawn. Each test case is a sequence of random appropriately typed values that can be used as input for the program under test.

Each member of the constructed pool of test cases has been validated against a fixed set of mutants. This validation procedure associates each test case with a corresponding mutation score that is related to the performance of the test case against a given fixed set of mutants. The higher the mutation score the more effective is a test case in killing mutants, so, in this way the score is a measure of the ability of the test case to kill mutants. This validation procedure has been simulated. The simulation generates random real numbers between 0.0 and 1.0; each number represents a mutation score associated with a test case.

The main idea is to construct subsets of these test cases, each subset to be considered as an individual of the genetic algorithm. Therefore the initial population of the genetic algorithm consists of subsets of test cases that are randomly selected from the pool of test cases. The simulation does this by grouping a number of mutation scores together; each group of mutation scores correspond to a subset of test cases.

Evolution of this population with the GA, will lead to subsets of test cases that are combined to give the maximum total mutation score. This is done by selecting the fitness function of an individual to be the sum of the mutation scores of the test cases divided by the length of the individual. That is, if MS_i is the mutation score of a test case i and the individual consists of L test cases the fitness Tf of this individual is given by:

$$Tf = \frac{\sum_{i=1}^{L} MS_i}{L} \tag{2}$$

The normalization of the fitness values, in order to get real values between 0.0 and 1.0, makes the fitness of individuals with different length comparable. As for the mutants, the fitness function is relatively easy to optimise. However, the presented here technique is a useful step towards the ultimate goal of achieving full co-evolution of both test set and mutant set.

Similarly with the GA for the mutants above, we use uniform crossover. This well-known crossover operator is applied to the individuals of the population.

Each test case is a gene of the individual and each individual is a set of test cases. The mutation operator refers to exchange of a test case of an individual with another test case randomly selected from the pool of test cases. The selection operator is the roulette-wheel selection that ensures that the probability of selection of an individual is proportional to its fitness.

Elitism is defined in the same way as for the previous GA in Section 2.1; a user defined number of high performance sets of test cases pass automatically to the next generation.

By applying this GA we work with a large set of test cases, and at the same time genetically select sets of test cases with higher fitness values. That is, higher ability in killing mutants and achieving high mutation scores. This method can automatically generate high performance test cases for an original program under test.

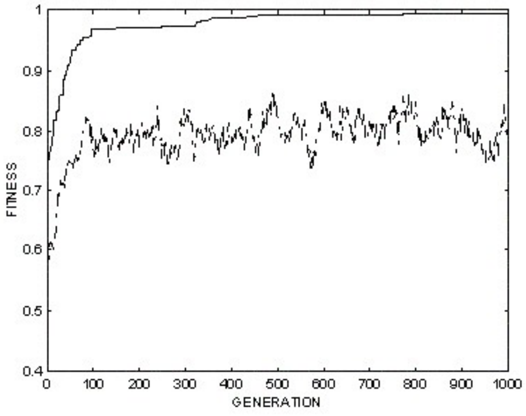

Fig. 1. GA performance for a standard set of parameter values used as a reference.

2.3 Co-evolution of Sets of Mutants and Sets of Test Cases

The concept of co-evolution in evolutionary computing is inspired by nature, where organisms that are ecologically intimate, for example, predators and prey, or hosts and parasites, influence each other's evolution. Co-evolution entails a change in the genetic composition of one species (or group) in response to a genetic change in another. This approach has been chosen to be part of our methodology for two main reasons:

1. **We achieve selective mutation**. Previous work has shown selection of mutation operators to be a productive way to reduce the (often infeasibly

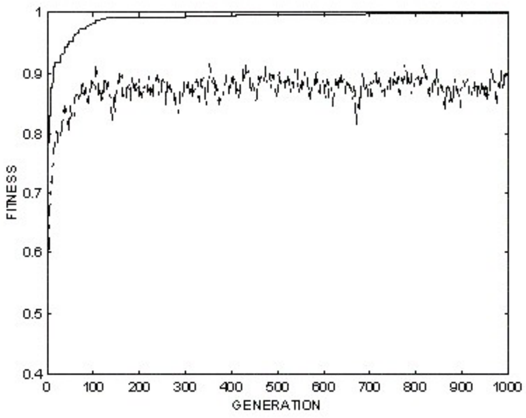

Fig. 2. Improved GA performance for higher elitism value, increased from 2 to 10.

large) number of mutants. However previous work requires that the mutation operators are selected *a priori*. Using evolution we are able to select individual mutants, tailored to the program under test, based upon their individual fitness.

2. **Our fitness function guarantees not to generate equivalent mutants**. The fitness function for mutants requires that at least one test case kills the mutant. Those mutants which are killed by no test cases, receive a very low value of fitness. This means that mutants are guaranteed to be non-equivalent. It may be that some *stubborn* (hard to kill) mutants which are not equivalent are also given low fitness scores. It is hoped that the robustness of the evolutionary search strategy will ensure that such stubborn mutants are rediscovered. That is, as the co-evolving test set improves its fitness, a point will be reached where the test set can kill the stubborn mutant. The authors believe that the ability to guarantee that all mutants are non-equivalent is an important goal. It justifies the 'aggressive' approach adopted here, because the presence of equivalent mutants makes mutation testing infeasible.

These two properties of our approach aim to address the two fundamental barriers to wider uptake of mutation testing, specifically:

1. Too many mutants are generated.
2. Too much effort is spent on identifying equivalent mutants.

In this paper the concept of co-evolution is based on the idea that two populations, that is the one consisting of individuals of mutants, and the one consisting of individuals of test cases, are evolved in parallel with a specifically designed GA. On each generation of the GA, the fitness of the individuals of these two

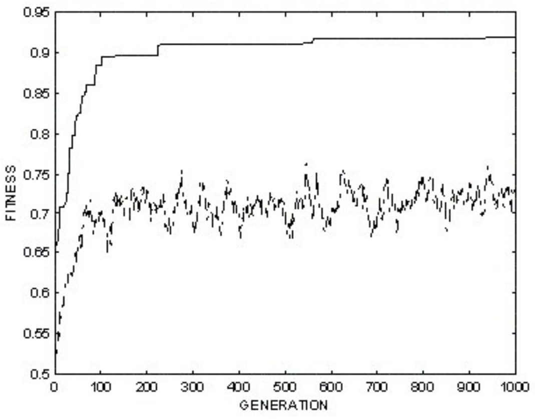

Fig. 3. Lower GA performance for higher individual's length, increased from 10 to 30.

populations are recalculated considering the alteration of the individuals of these two populations.

In other words the score of each mutant is re-evaluated with respect to the present population of the test cases, and on the other hand the score of each test case is re-evaluated with respect to the present population of mutants. Co-evolution of these two populations is expected to lead to individuals consisting of mutants that are very difficult to be killed and individuals consisting of test cases that are capable of killing high quality mutants.

3 Results and Discussions

This section presents simulated results for the GA that evolves sets of mutants. A pool of N mutants was generated and evaluated. Figure 1 shows the performance of the GA for N=10,000, constant population size POP of 100, individual's length L=10 (10 mutants in each set), crossover probability Pc=1.0, mutation probability Pm=0.05, and elitism E=2. The GA ran for 1000 generations and the results presented are averaged over 10 runs for each GA.

The solid line shows the evolution of the fitness of the fittest individual of each generation, whereas the dotted line represents the evolution of the mean fitness of each generation.

By increasing the value of elitism from 2 to 10, leaving the values of the rest of the parameters the same, we obtain the results shown in Figure 2.

We observe that in this case the algorithm converges faster, and achieves a higher value of best fitness. This result introduces the idea of how elitism, or similar operators, can be used in the future as a kind of memory to improve the performance of the algorithm.

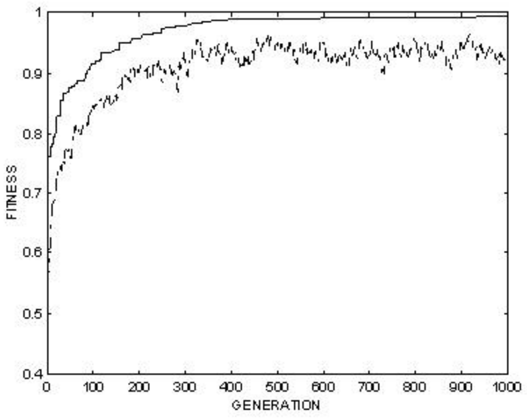

Fig. 4. Higher performance for the mean fitness of each generation when the mutation probability has decreased from 0.05 to 0.01.

Figure 3 shows the performance of the GA when the length of each individual has increased to 30 genes. The values of the rest of the parameters are the same as for Figure 1. In this case the algorithm converges more slowly and does not perform as well. It appears that the number of genes forming an individual is a critical parameter for the performance of the GA.

Figure 4 illustrates the performance of the GA when the mutation probability has decreased from 5% to 1%, while the values of the other parameters are the same as in Figure 1. The algorithm achieves an increase in the effectiveness of the average fitness of each generation.

4 Future Work

In future work, more simulated results from all the GAs will be presented. These results will be compared with those obtained from a real mutation testing tool, which is currently under development.

In addition more results will be presented concerning the performance of the GAs and the influence of GA parameters (population size, crossover probability, choice between one-point crossover and uniform crossover, mutation probability, length, selection scheme, elitism) on that performance.

Moreover elitism will be expanded to offer a kind of memory of the best individuals. The influence of this memory on GA performance will be further analysed.

Additionally a comparative analysis with selective mutation and mutation sampling will be presented.

Finally the technique used for elimination of equivalent mutants will be analysed in order to examine if this technique results in killing also stubborn mutants.

5 Conclusion

In this paper we addressed two main problems of mutation testing. First, the problem of equivalent mutant detection; we showed how to design a fitness function in such a way that possible equivalent mutants can be detected. Second, the problem of the large number of mutants produced; we showed how to select effective mutants and test cases of an original program. We approached these problems as search problems using probabilistic, meta-heuristic algorithms such as Genetic Algorithms and co-evolution.

Our main motivation was to overcome the twin difficulties of the infeasibly large number of generatable new mutants and the problem of weeding out equivalent mutants. The proposed method does not reject mutation operators like the method of selective mutation, nor does it decrease the number of mutants by selecting a random sample of the mutants. Instead the proposed method generates a pool of mutants that can be generated from an original program, after the application of all the mutation operators. As a second step the method utilizes a GA that evolves subsets of mutants and therefore genetically rejects irrelevant and low performance mutants.

In this way we overcome one of the principal disadvantages of selective mutation; false generalisation for the rejection of mutation operators.

In the same manner genetic evolution on sets of test cases is proposed as a method for increasing their testing ability according to their adequacy score.

Finally these two methods are incorporated in an algorithm for the co-evolution of both mutants and test cases in parallel. These two populations compete with each other and therefore better results are expected.

References

1. Myers, G.J.: The Art of Software Testing. Wiley - Interscience, New York (1979)
2. Pressman, R.: Software Engineering: A Practitioner's Approach. 3rd edn. McGraw-Hill Book Company Europe, Maidenhead, Berkshire, England, UK. (1992) European adaptation (1994). Adapted by Darrel Ince. ISBN 0-07-707936-1.
3. Hamlet, R.G.: Testing programs with the aid of a compiler. IEEE Transactions on Software Engineering **3** (1977) 279–290
4. Bottaci, L., Mresa, E.S.: Efficiency of mutation operators and selective mutation strategies: An empirical study. Software Testing, Verification and Reliability **9** (1999) 205–232
5. Budd, T.A.: Mutation analysis: Ideas, examples, problems and prospects. In: Proceedings of the Summer School on Computer Program Testing, Sogesta (1981) 129–148
6. Howden, W.E.: Weak mutation testing and completeness of test sets. IEEE Transactions on Software Engineering **8** (1982) 371–379
7. Woodward, M.R., Halewood, K.: From weak to strong, dead or alive? an analysis of some mutation testing issues. In: Proceedings of the Second Workshop on Software Testing, Verification, and Analysis, Banff, Canada (1988)
8. Untch, R.H., Offutt, A.J., Harrold, M.J.: Mutation analysis using mutant schemata. In Ostrand, T., Weyuker, E., eds.: Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA). (1993) 139–148

9. Byoungju, C., Mathur, A.P.: High-performance mutation testing. The Journal of Systems and Software **20** (1993) 135–152
10. Offutt, A.J., Untch, R.: Mutation 2000: Uniting the orthogonal. In Wong, W.E., ed.: Mutation Testing for the New Century (proceedings of Mutation 2000), San Jose, California, USA, Kluwer (2001) 45–55
11. Baldwin, D., Sayward, F.: Heuristics for determining equivalence of program mutations. Research Report 276, Department of Computer Science, Yale University (1979)
12. Hierons, R.M., Harman, M., Danicic, S.: Using program slicing to assist in the detection of equivalent mutants. Software Testing, Verification and Reliability **9** (1999) 233–262
13. Offutt, A.J., Craft, W.M.: Using compiler optimization techniques to detect equivalent mutants. Software Testing, Verification and Reliability **4** (1994) 131–154
14. Offutt, A.J., Pan, J.: Detecting equivalent mutants and the feasible path problem. In: Annual Conference on Computer Assurance (COMPASS 96), IEEE Computer Society Press, Gaithersburg, MD (1996) 224–236
15. Offutt, A.J., Pan, J.: Automatically detecting equivalent mutants and infeasible paths. Software Testing, Verification, and Reliability **7** (1997) 165–192
16. Bottaci, L.: Instrumenting programs with flag variables for test data search by genetic algorithms. In Langdon, W.B., Cantú-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M.A., Schultz, A.C., Miller, J.F., Burke, E., Jonoska, N., eds.: GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, New York, Morgan Kaufmann Publishers (2002) 1337–1342
17. Bottaci, L.: Predicate expression cost functions to guide evolutionary search for test data. In Cantú-Paz, E., Foster, J.A., Deb, K., Davis, D., Roy, R., O'Reilly, U.M., Beyer, H.G., Standish, R., Kendall, G., Wilson, S., Harman, M., Wegener, J., Dasgupta, D., Potter, M.A., Schultz, A.C., Dowsland, K., Jonoska, N., Miller, J., eds.: Genetic and Evolutionary Computation – GECCO-2003. Volume 2724 of LNCS., Chicago, Springer-Verlag (2003) 2455–2464
18. Ferguson, R., Korel, B.: The chaining approach for software test data generation. ACM Transactions on Software Engineering and Methodology **5** (1996) 63–86
19. Jones, B., Sthamer, H.H., Eyres, D.: Automatic structural testing using genetic algorithms. The Software Engineering Journal **11** (1996) 299–306
20. Jones, B.F., Eyres, D.E., Sthamer, H.H.: A strategy for using genetic algorithms to automate branch and fault-based testing. The Computer Journal **41** (1998) 98–107
21. Michael, C., McGraw, G., Schatz, M.: Generating software test data by evolution. IEEE Transactions on Software Engineering (2001) 1085–1110
22. Mueller, F., Wegener, J.: A comparison of static analysis and evolutionary testing for the verification of timing constraints. In: 4th IEEE Real-Time Technology and Applications Symposium (RTAS '98), Washington - Brussels - Tokyo, IEEE (1998) 144–154
23. Pargas, R.P., Harrold, M.J., Peck, R.R.: Test-data generation using genetic algorithms. The Journal of Software Testing, Verification and Reliability **9** (1999) 263–282
24. Pohlheim, H., Wegener, J.: Testing the temporal behavior of real-time software modules using extended evolutionary algorithms. In Banzhaf, W., Daida, J., Eiben, A.E., Garzon, M.H., Honavar, V., Jakiela, M., Smith, R.E., eds.: Proceedings of the Genetic and Evolutionary Computation Conference. Volume 2., Orlando, Florida, USA, Morgan Kaufmann (1999) 1795

25. Schultz, A., Grefenstette, J., Jong, K.: Test and evaluation by genetic algorithms. IEEE Expert **8** (1993) 9–14
26. Tracey, N., Clark, J., Mander, K.: The way forward for unifying dynamic test-case generation: The optimisation-based approach. In: International Workshop on Dependable Computing and Its Applications (DCIA), IFIP (1998) 169–180
27. Wegener, J., Grimm, K., Grochtmann, M., Sthamer, H., Jones, B.F.: Systematic testing of real-time systems. In: 4th International Conference on Software Testing Analysis and Review (EuroSTAR 96). (1996)
28. Wegener, J., Sthamer, H., Jones, B.F., Eyres, D.E.: Testing real-time systems using genetic algorithms. Software Quality **6** (1997) 127–135
29. McMinn, P.: A survey of evolutionary testing. (Software Testing, Verification and Reliability) To appear.
30. Kirsopp, C., Shepperd, M., Hart, J.: Search heuristics, case-based reasoning and software project effort prediction. In Langdon, W.B., Cantú-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M.A., Schultz, A.C., Miller, J.F., Burke, E., Jonoska, N., eds.: GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, New York, Morgan Kaufmann Publishers (2002) 1367–1374
31. DeMillo, R.A., Offutt, A.J.: Experimental results from an automatic test generator. acm Transactions of Software Engineering and Methodology **2** (1993) 109–127