EuroSTAR 2001

Application Fields for Evolutionary Testing

Joachim Wegener, Harmen Sthamer, and André Baresel
DaimlerChrysler AG, Research and Technology, Alt-Moabit 96a, D-10559 Berlin, Germany
Joachim.Wegener@DaimlerChrysler.com
Harmen.Sthamer@DaimlerChrysler.com
Andre.Baresel@DaimlerChrysler.com

Abstract

Dynamic testing is the most important analytic quality measure within the development of software-based systems. Frequently, 30%-50% of the total development expenses go to the testing of these systems. Test case design is the decisive test activity for test quality, since it determines type and scope of the test. For most test objectives, test case design is difficult to automate. Therefore, it has to be performed manually, which in turn increases costs and expenses for the test. Furthermore, the quality of the determined test case is strongly influenced by the performance of the tester.

A promising approach to automate test case design for various test objectives is Evolutionary Test. Evolutionary testing refers to the use of meta-heuristic search techniques for test data generation. Whenever a test objective may be expressed numerically, the deployment of evolutionary testing for the automation of test case design is possible. The appropriate formalisation of the test objective is the key to success. This paper intends to give an overview of the processing of different test objectives in order to achieve their automation by evolutionary testing.

0 Introduction

Of all testing activities, test case design has most influence on the overall test quality since it defines type and scope of the test. For this reason, a variety of different test methods were developed in the last century, intended to support systematic test case determination. The procedures may be divided into functional, structural, statistical, and diversifying procedures. All procedures clearly emphasise the testing of logical program behaviour. At present, comprehensive test automation is only possible for statistical test methods, since formal specification techniques have not yet gained widespread acceptance within industry. For the automation of statistical testing, the availability of an operational profile and a test oracle is prerequisite. These preconditions are not available for the vast majority of the systems. Therefore, for most test objectives, test case design is difficult to automate and has to be performed manually:

- for functional testing, the generation of test cases is usually impossible because no formal specifications are applied in industrial practice,
- structural and mutation testing are difficult to automate due to the limits of symbolic execution, and
- for testing non-functional constraints like temporal behaviour, safety and robustness of systems no specialized methods and tools exist.

To increase the effectiveness and efficiency of the test and thus reduce the overall development costs for software-based systems, we require a test that is systematic and extensively automatable. While functional test case design can be automated to a large extent using new tools such as the CTE XL [Lehmann and Wegener, 2000], evolutionary testing is a promising approach for fully automating test case design for the other aspects mentioned above. The Evolutionary Test can be used to generate test cases for structural testing, and it facilitates the automation of testing non-functional constraints.

For evolutionary testing, the test case design has to be transformed into an optimisation problem that in turn is solved with meta-heuristic search techniques, such as evolutionary algorithms or simulated annealing. The input domain of the system under test represents the search space in which test data fulfilling the test objectives under consideration is searched for. The Evolutionary Test is applicable in general because it adapts itself to the system under test. Effectiveness and efficiency of the test process can be clearly improved by Evolutionary Tests. This has been successfully proved in several case studies. Evolutionary Tests thus contribute to quality improvement as well as to the reduction of development costs.

The first chapter introduces the basic principles of the Evolutionary Test. The second chapter discusses its use for structural testing. The following chapter describes the use of evolutionary testing for the examination of non-functional constraints, namely temporal behaviour testing, safety testing, and robustness testing. The paper concludes with a summary of the most important results and an outlook on future work.

1 Evolutionary Testing

Evolutionary testing is characterised by the use of meta-heuristic search methods for test case generation. To achieve this, the considered test aim is transformed into an optimisation problem. The input domain of the test object forms the search space in which one searches for test data that fulfils the respective test aim. Due to the non-linearity of software (if-statements, loops etc.) the conversion of test problems to optimisation tasks mostly results in complex, discontinuous, and non-linear search spaces. Neighbourhood search methods like hill climbing are not suitable in such cases. Therefore, meta-heuristic search methods are employed, e.g. evolutionary algorithms, simulated annealing, or taboo search. In our work, evolutionary algorithms are used to generate test data because their robustness and suitability for the solution of different test tasks has already been proven in previous work, e.g. [Jones et al., 1998] and [Wegener et al., 1999].

1.1 A Brief Introduction to Evolutionary Algorithms

Evolutionary algorithms represent a class of adaptive search techniques and procedures based on the processes of natural genetics and Darwin's theory of biological evolution. They are characterised by an iterative procedure and work in parallel on a number of potential solutions – the population of individuals. Permissible solution values for the variables of the optimisation problem are encoded in each individual.

The fundamental concept of evolutionary algorithms is to evolve successive generations of increasingly better combinations of those parameters that significantly affect the overall performance of a design. Starting with a selection of good individuals, the evolutionary algorithm attempts to achieve the optimum solution by random exchange of information between increasingly fit samples (recombination) and introduction of a probability of independent random change (mutation). The adaptation of the evolutionary algorithm is achieved by selection and reinsertion procedures which are based on the individuals' fitness values. Selection procedures control which individuals are selected for reproduction. The reinsertion strategy determines how many and which individuals are taken from the parent and the offspring population to form the next generation.

The fitness value is a numerical value that expresses the performance of an individual with regard to the current optimum, so that different individuals can be compared. The notion of fitness is fundamental to the application of evolutionary algorithms; the degree of success in using them depends critically on the definition of a fitness function that changes neither too rapidly nor too slowly with the design parameters. The fitness function must guarantee that individuals can be differentiated according to their suitability for solving the optimisation problem.

Fig. 1 provides an overview of a typical procedure for evolutionary algorithms. First, a population of guesses on the solution of a problem is initialised, usually at random. Each individual within the population is evaluated by calculating its fitness. This will usually result in a spread of solutions ranging in fitness from very poor to good. The remainder of the algorithm is iterated until the optimum is achieved, or another stopping condition is fulfilled. Pairs of individuals are selected from the population according to the pre-defined selection strategy, and combined in some way to produce a new guess analogously to biological reproduction. Additionally, mutation is applied. The new individuals are evaluated for their fitness, and survivors into the next generation are chosen from parents and offspring, often according to fitness. It is important, however, to maintain diversity in the population to prevent premature convergence to a sub-optimal solution.

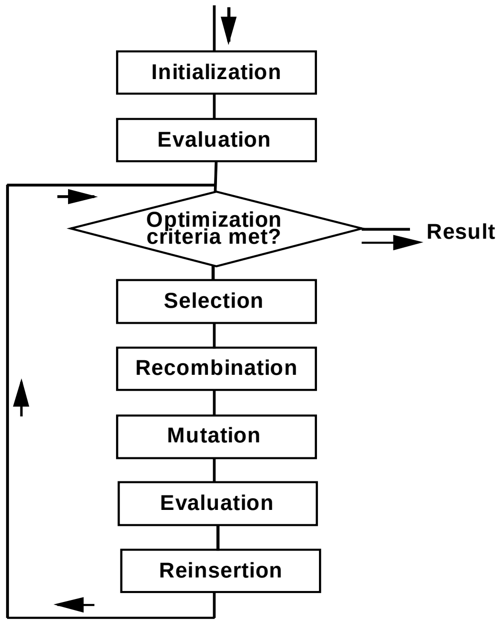

Figure 1: Evolutionary Algorithms

1.2 Application to Software Testing

In order to automate software tests with the aid of evolutionary algorithms, the test aim must itself be transformed into an optimisation task. For this, a numeric representation of the test aim is necessary, from which a suitable fitness function for the evaluation of the generated test data can be derived. Each generated individual represents a test datum for the system under test. Depending on which test aim is pursued, different fitness functions emerge for test data evaluation. If an appropriate fitness function can be defined, then the Evolutionary Test proceeds as follows.

The initial population is usually generated at random. In principle, if test data has been obtained by a previous systematic test, this could also be used as initial population [Wegener et al., 1996]. The Evolutionary Test could thus benefit from the tester's knowledge of the system under test. Each individual of the population represents a test datum with which the test object is executed. For each test datum the execution is monitored and the fitness value is determined for the corresponding individual. Next, population members are selected with regard to their fitness and subjected to combination and mutation processes to generate new offspring. It is important to ensure that the test data generated is in the input domain of the test object. Offspring individuals are then also evaluated by executing the corresponding test data. Combining offspring and parent individuals, according to survival procedures, forms a new population. From here on, the process repeats itself, starting with selection, until the test objective is fulfilled or another given stopping condition is reached (see Fig. 2).

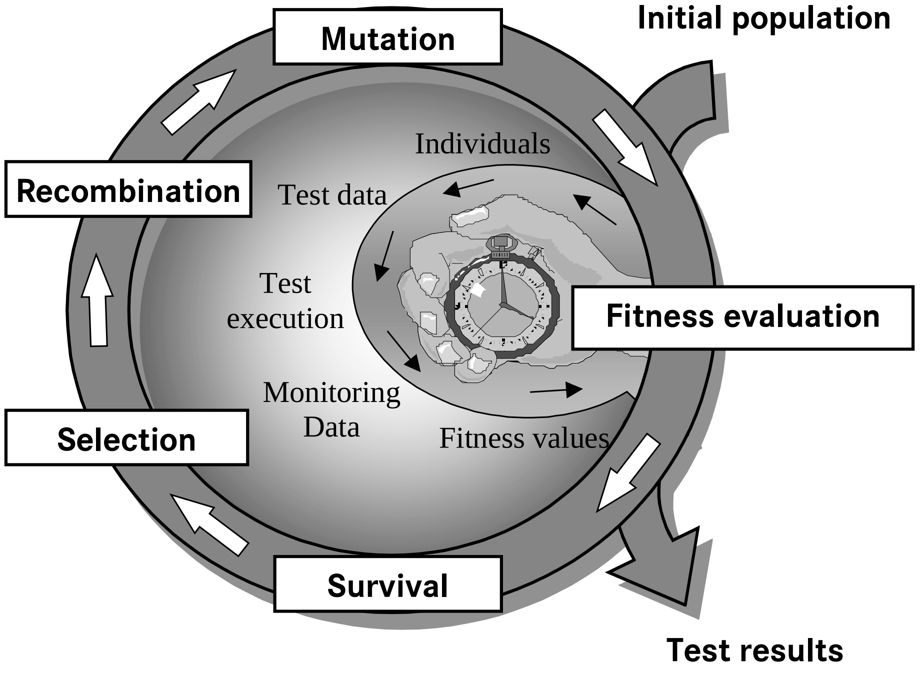

Figure 2: Evolutionary Test

2 Test Case Generation for Structural Testing

Structural testing is widespread in industrial practice and stipulated in many software-development standards. Common examples are statement, branch, and condition testing. The aim of applying evolutionary testing to structural testing is the generation of a quantity of test data, leading to the highest possible coverage of the selected structural test criterion.

Structural testing methods can be divided into four main categories, depending on the control-flow graph and the required purpose of the test:

- node-oriented methods,
- path-oriented methods,
- node-path-oriented methods, and
- node-node-oriented methods.

Node-oriented methods require the execution of specific nodes in the control-flow graph. Statement testing and condition testing are the best known methods that fall into this category. Path-oriented methods require the execution of certain paths in the control-flow graph, e.g. path testing. Node-path-oriented methods require the achievement of a specific node and from this node the execution of a specific path through the control-flow graph. The branch test is the simplest example for node-path-oriented methods. LCSAJ (linear code sequence and jump) and all-defuse-chains also belong to the group of node-path-oriented methods. Node-node-oriented methods require the execution of several nodes of the control-flow graph in a pre-determined sequence without specifying a concrete path. Most data-flow oriented methods like all-defs and all-uses fit into this category.

In order to apply evolutionary testing to the automation of structural testing, the test is split up into partial aims. The identification of the partial aims is based on the control-flow graph of the program under test. Each partial aim represents a program structure that needs to be executed to achieve full coverage, e.g. a statement, a branch, or a condition with its logical values. For each partial aim an individual fitness function is formulated and a separate optimisation is performed to search for a test datum executing the partial aim. The set of test data found for the partial aims then serves as the test data set for the coverage of the structure test criterion.

2.1 Fitness Functions

The fitness function definitions for the partial aims differ for the four categories of structural testing methods.

For node-oriented methods, the fitness functions of the partial aims are made up of two components: the distance and the approximation level. The distance specifies for a branching node how far away an individual is from executing the branching conditions in the desired manner (see [Sthamer, 1996], [Jones et al., 1998], and [Tracey et al., 1998]). For example, if a branching condition *x==y* needs to be evaluated as *True*, then the fitness function may be defined as $|x-y|$ (provided that the fitness values are minimised during the optimisation) or as hamming distance. The approximation level supplies a figure for an individual that gives the number of branching nodes lying between the nodes covered by the individual and the target node ([Wegener et al., 2001], and [Baresel, 2000]). For condition tests the fitness evaluation needs to be slightly extended. The evaluation of the atomic predicates in the target nodes has to be included. The evaluation of the atomic predicates takes place in the same way as for the distance calculations in the branching conditions. For compound predicates the single distances are added and normalised.

Establishing the fitness function for path-oriented testing methods is much simpler than for node-oriented methods because the execution of a certain path through the control-flow graph forms the partial aim for the Evolutionary Test. The program path covered by an individual is compared with the program path specified as a partial aim. Thereby, the more nodes match, the higher is the fitness an individual can obtain. The fitness evaluation is supplemented by the calculation of the distances to the target path in the branching nodes in which the program path covered by the individual deviates from the target path.

The partial aims for node-path-oriented structural criteria comprise two requirements that need to be included in the evaluation of the generated individuals. The attainment of a specific node is required on the one hand, and on the other hand a path that begins with this node has to be covered. Accordingly, the fitness evaluation of the individuals has to represent both these components. The fitness function can be based on the fitness functions for node-oriented and path-oriented methods. Fitness calculations for individuals who do not reach the target node are carried out in the same manner as for the node-oriented methods. For individuals who reach the target node the mentioned fitness calculations for path-oriented methods are additionally applied in order to guide the search into the direction of the desired path.

Fitness calculations for node-node-oriented methods also take place in two stages. After the execution of the first target node, the second target node has to be covered, without a path specified through the control-flow graph. The approximation of an individual to the first target node can be evaluated in the same manner as for node-oriented methods. For all individuals executing the first target node an approximation to the second node is added. This is also calculated using the fitness function for node-oriented methods.

A detailed definition of the fitness functions can be found in [Wegener et al., 2001] and [Baresel, 2000]. Enhancements to apply evolutionary structural testing to mutation testing are described in [Bottaci, 2001] and [Tracey et al., 2000].

2.2 Test Results

The Evolutionary Test has already been applied in various tests of real-world examples for the automatic generation of test data with excellent results – for all test objects, complete branch coverage was achieved, and for most test objects evolutionary testing performed notably better than random testing. Evolutionary testing achieved higher branch coverage than random testing, and also needed much fewer test data to be generated (Table 1).

Table 1 shows a selection of examined test objects from different application fields with varying complexity. Complexity metrics used are the number of code lines (LOC), the number of program branches (NB) corresponding to the number of partial aims for the test, the cyclomatic complexity (CC), the maximum nesting level (NL), and Myer's interval (MI). Furthermore, the table shows the degree of branch coverage (COV) reached for each test object by random testing (RT) and evolutionary testing (ET), and the number of test data sets generated (GTD, in thousands). The branch coverage reached varies between 47 % and 100 % for random testing, whereas evolutionary testing achieves full coverage for all examples. The Netflow() function contains one infeasible program branch. Therefore, the highest possible coverage is around 99%.

For the more complex test objects, the time for the evolutionary tests varied between 160 and 250 seconds. This results in a mean processing time of 4 to 6 seconds per partial aim.

Test object	**LOC**	**NB**	**CC**	**NL**	**MI**	**RT**		**ET**	
						COV	**GTD**	**COV**	**GTD**
Atof()	64	57	16	2	27	47	1251	**100**	35
Is_line_covered_by_rectangle()	71	24	8	4	6	**100**	1	**100**	1
Is_point_located_in_rectangle()	5	5	2	1	3	**100**	1	**100**	1
Search_field()	108	37	13	2	10	92	920	**100**	14
Netflow()	154	153	34	7	40	97	1091	**99**	41
Complex_Flow()	42	41	13	2	10	98	471	**100**	29
Classify_Triangle()	35	38	14	2	7	91	200	**100**	17

Table 1: Complexity Measures and Branch Coverage Reached for Different Test Objects

3 Test Case Generation for Examination of Non-Functional Constraints

For a series of systems, the correct operating depends not only on the logical processing results but also on adhering to specified non-functional properties. Real-time systems and interactive systems, for instance, have to react within a certain time range. For safety-relevant systems, it has to be ensured that defined safety constraints are violated under no circumstances by the system. Almost all developed systems have to be robust against unexpected inputs.

3.1 Testing Temporal Behaviour

Most embedded systems and most interactive systems are subject to temporal requirements. This is due to reasons of operational comfort, e.g. short reaction times of the system to user commands, or due to requirements of technical processes that are controlled by the system. Therefore, the developed systems have to be thoroughly tested not only with regard to their functional behaviour, but also in order to detect existing deficiencies in temporal behaviour.

Existing test methods are unsuitable for the examination of temporal correctness. Even for an experienced tester it is virtually impossible to find the most important input situations relevant for a thorough examination of temporal behaviour by analysing and testing complex systems manually. However, evolutionary testing has already proved to be a promising approach for testing the temporal behaviour of real-time and embedded systems, e.g. [Mueller and Wegener, 1998], [Puschner and Nossal, 1998], [Wegener and Grochtmann, 1998] and [Gross et al., 2000]. When testing the temporal behaviour of systems the objective is to check whether input situations exist for which the system violates its specified timing constraints. Usually, a violation occurs because outputs are produced too early or their computation takes too long. The task of the tester and, therefore, of the Evolutionary Test is to find input situations with especially long or short execution times in order to check whether a temporal error can be produced.

When using evolutionary testing for determining the shortest and longest execution times of test objects, the execution time is measured for every test datum. The fitness evaluation of the generated individuals is based on the execution times measured for the corresponding test data. If one searches for long execution times, individuals with long execution times obtain high fitness values. Conversely, when searching for short execution times, individuals with short execution times obtain high fitness values. Individuals with long or short execution times are selected depending on the test objective and combined in order to obtain test data with even longer or shorter execution times. The test is terminated if an error in the temporal behaviour is detected or a specified termination criterion has been reached. If a violation of the system's predetermined temporal limits has been detected, the test was successful and the system has to be corrected. Evolutionary testing enables a fully automated search for extreme execution times. The test especially benefits from the fact that test evaluation concerning temporal behaviour is usually trivial. Contrary to logical behaviour, the same timing constraints apply to large numbers of input situations.

3.1.1 Test Results

Previous work has shown that evolutionary testing always achieved better results than random testing when testing temporal behaviour, e.g. [Wegener and Grochtmann, 1998]. For embedded real-time systems the comparison with static analyses has also confirmed that the extreme execution times determined by the Evolutionary Test represent realistic estimations of the longest and shortest execution times [Mueller and Wegener, 1998]. Compared to systematic tests, the Evolutionary Test has also attained convincing results (see [Wegener et al., 1999]). The following results were achieved during the first application of evolutionary testing for the testing of a new engine control system for six- and eight-cylinder blocks.

The engine control system contains several tasks that have to fulfil timing constraints. Each task is a test object and has been tested for its worst-case execution time by the developers using systematic testing. The test cases for testing the temporal behaviour, defined by the developers, are based on the functional specification of the system as well as on the internal structures of the tasks. For each task the developer tests achieved full branch coverage. Evolutionary testing was used to verify these results. The tests were performed on the target processor later used in the vehicles. The execution times have been determined using hardware timers of the target environment.

The results for six of the tasks (M1 to M6) are shown in Figure 3. The figure shows the longest execution times determined by the developers with systematic testing (DT) in comparison to the results achieved by evolutionary testing (ET). Additionally, the results for random testing are shown (RT). The results of the developer tests are set to be 100 %. The execution times achieved, measured in µs, are shown directly in the bars. The size of the tasks varied from 39 LOC (lines of code) to 119 LOC, the number of input parameters from 9 to 32.

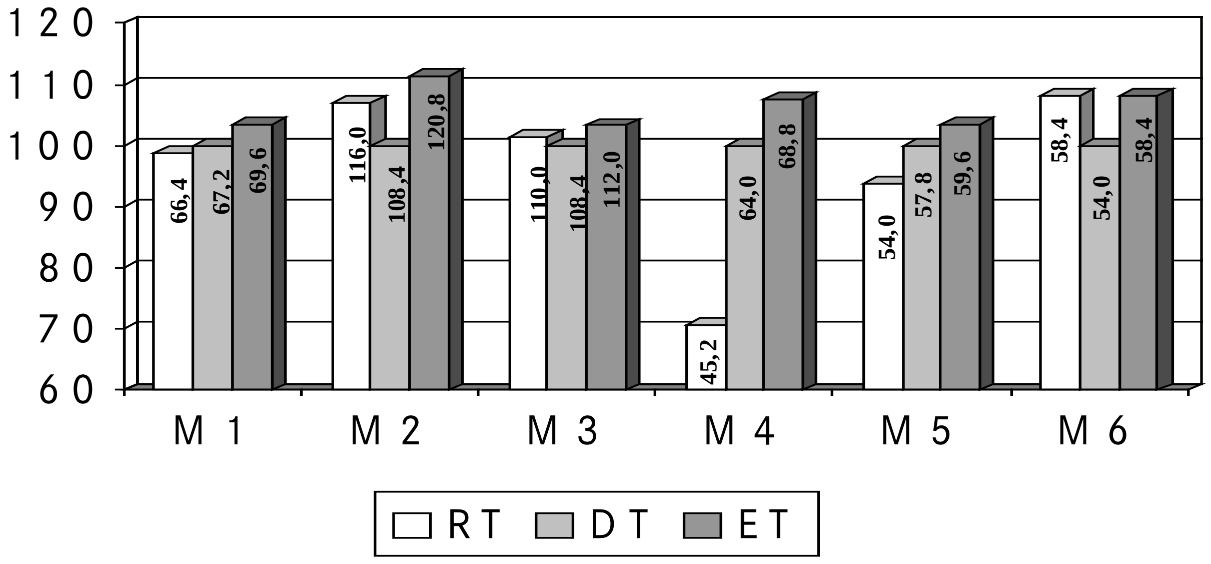

Figure 3: Results for the Engine Control System Tasks

The comparison of the results shows that evolutionary testing found the longest execution times for all the given tasks among these three testing methods. The developer tests never reached the longest execution time. In three cases the results of the developer tests are even worse than those of random testing. For the other three tasks the results are better than those of the random test. The excellent performance of the Evolutionary Test in comparison to the developer tests shows the effectiveness of the Evolutionary Test. The results are especially astonishing, because evolutionary testing treats the software as black boxes whereas the developers are familiar with function and structure of their system. An explanation might be the use of system calls of which the effects on the temporal behaviour can only be rated with difficulty by the developers.

3.2 Safety Testing

Our work on the application of evolutionary testing for testing safety properties is just beginning. It follows the work of Tracey et al. [1998]. Within the context of safety analyses for software-based systems (e.g. fault-tree analysis, and software-hazard analysis) indispensable safety requirements for the system components are derived from such system behaviour that has to be absolutely avoided. If a violation of the specified safety requirements is possible the system is not safe. Consequently, the aim of the test is to find input situations that lead to a violation of the safety requirements. If such an input situation can be found the system is not safe and has to be corrected.

The fitness evaluation when applying the Evolutionary Test to safety tests is similar to the fitness evaluation of structural testing. However, the fitness function is not based on the branch predicates of the program, but on the pre- and post-conditions that have been specified for the single components (e.g. [Tracey et al., 1998]). For example, if an output signal *speed* of a component is not allowed to become negative, the fitness values of the individuals can be set according to every produced output value for *speed.* Individuals who generate a small value for *speed* obtain a higher fitness value than individuals producing high values for *speed.* If the Evolutionary Test is able to find an individual who obtains a negative value for *speed*, it is proof of a violation of the safety requirements.

In order to achieve a complete automation of safety testing, we are currently working on an integration of the Evolutionary Test with Time Partition Testing [Lehmann, 2000] for the system and integration test of embedded systems.

3.3 Robustness Testing

Works referring to the automation of robustness tests with evolutionary algorithms come from Boden and Martino [1996], and Schultz et al. [1993].

Boden and Martino concentrate their tests on the error treatment routines of an operating system API. The fitness function attempts to assess sequences of operating system calls.

Schultz et al. test error tolerance mechanisms of an autonomous vehicle. For this, Schultz et. al. search for error scenarios, which either lead to a failure of the autonomous controller with already a small amount of errors, or, which lead despite a large amount of errors to the successful completion of the mission. The fitness function consists of 1/(fault_activity * score), with fault_activity referring to the amount of occurring errors, and score assessing the performance of the controller. A high value for score represents a good controller performance, whereas low values imply the failure of the controller. If the fitness function is being minimised, scenarios will be obtained displaying a good controller performance despite many errors. If the fitness function is being maximised, scenarios evolve leading to a malfunction of the controller with a low amount of errors. Both Boden and Martino, and Schultz report on positive test results which have lead to an improvement of the respective products.

4 Summary and Future Work

The thorough test of software-based systems could include a number of demanding testing tasks. The test case design for various test objectives is difficult to master on the basis of conventional function-oriented and structure-oriented testing methods. Moreover, automation of test case design is problematic. Usually, test cases have to be defined manually.

Evolutionary Testing is a promising approach for fully automate complex testing tasks. As described in this paper, it enables the complete automation for structural test case design, for testing temporal behaviour with regard to its exceeding or falling below the specified timing constraints, and for the testing of safety properties and program robustness. Evolutionary testing has already produced very good results in all these application fields. Effectiveness and efficiency of the test process can be clearly improved by Evolutionary Tests. Evolutionary Tests thus contribute to quality improvement and to reduction of development costs.

Due to the full automation of the Evolutionary Testing, the system is tested with a large number of different input situations. In most cases, more than several thousand test data sets are generated and executed within only a few minutes. The prerequisites for the application of Evolutionary Tests are extremely few. Only an interface specification of the system under test is needed to guarantee the generation of valid input values. For structural testing the source code of the test object is also required.

Current work on evolutionary structural tests concentrates on the assessment of the testability of programs on the basis of statically determinable software metrics. By using appropriate information it is possible to select the best suitable evolutionary algorithms for the test, and also to apply program transformations that improve the testability.

In addition we are looking at investigating the application of evolutionary structural tests for testing the temporal behaviour of systems. The idea is to pre-determine program paths as test aim for the evolutionary structural test which have been identified as worst-case execution time paths by means of static analyses (e.g. [Mueller, 1997], [Puschner and Vrchoticky, 1997]). If a test datum can be found that executes the path we can be sure that this is the longest execution time possible to obtain. Due to pessimistic assumptions in static analyses the path will usually not be executable. However, the pre-definition of these paths can lead to a very interesting concentration on paths with long execution times.

5 References

Baresel, A.: *Automatisierung von Strukturtests mit evolutionären Algorithmen (Automation of Structural Testing using Evolutionary Algorithms)*. Diploma Thesis, Humboldt University, Berlin, Germany, 2000.

Boden, E. and Martino, G.: *Testing Software Using Order-based Genetic Algorithms*. Proceedings of the 1st Conference on Genetic Programming, Stanford University, USA, 1996, pp. 461 - 466.

Bottaci, L.: *A Genetic Algorithm Fitness Function for Mutation Testing*, Workshop on Software Engineering using Metaheuristic Innovative Algorithms (SEMINAL) in Proc. 23rd International Conference on Software Engineering (ICSE), Toronto, Canada, May 2001, pp. 3 - 7.

Gross, H.-G., Jones, B. and Eyres, D.: *Structural performance measure of Evolutionary Testing applied to worst-case timing of real-time systems*. IEE Proc.-Softw., Vol. 147, No. 2, April 2000, pp. 25 – 30.

Jones, B. F., Eyres, D. E. and Sthamer, H.-H.: *A Strategy for using Genetic Algorithms to Automate Branch and Fault-based Testing*. The Computer Journal, Vol. 41, No. 2, 1998.

Lehmann, E.: *Time Partition Testing: A Method for Testing Dynamic Functional Behaviour*. Proceedings of TEST2000, London, Great Britain, May 2000.

Lehmann, E.; Wegener, J.: *Test Case Design by Means of the CTE XL*. Proceedings of the 8th European International Conference on Software Testing, Analysis & Review (EuroSTAR 2000), Kopenhagen, Denmark, December 2000.

Mueller, F.: *Generalizing Timing Predictions to Set-Associative Caches*. Proc. EuroMicro Workshop on Real-Time Systems, Jun 1997, pp. 64-71.

Mueller, F.; Wegener, J.: *A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints*. Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium, Denver, USA, June 1998.

Puschner, P. and Nossal, R.: *Testing the Results of Static Worst-Case Execution-Time Analysis*. Proc. 19th Real-Time Systems Symposium, 1998, pp. 134 - 143.

Puschner, P. and Vrchoticky, A.: *Problems in Static Worst-Case Execution Time Analysis*. Proceedings of the 9th ITG/GI-Conference Measurement, Modeling and Evaluation of Computational and Communication Systems, 1997, pp. 18 - 25.

Schultz, A. C., Grefenstette, J. J. and De Jong, K. A.: *Test and Evaluation by Genetic Algorithms*. IEEE Expert 8(5), 1993, pp. 9 – 14.

Sthamer, H.-H.: *The Automatic Generation of Software Test Data Using Genetic Algorithms*. PhD Thesis, University of Glamorgan, Pontyprid, Wales, Great Britain, 1996.

Tracey, N., Clark, J., Mander, K. and McDermid, J.: *An Automated Framework for Structural Test-Data Generation*. Proceedings of the 13th IEEE Conference on Automated Software Engineering, Hawaii, USA 1998.

Tracey, N.; Clark, J.; McDermid, J. and Mander, K.: *A Search Based Automated Test-Data Generation Framework for High-Integrity Systems.* Journal of Software Practice and Experience, January 2000.

Wegener, J., Grochtmann, M.: *Verifying Timing Constraints of Real-Time Systems by means of Evolutionary Testing*. Real-Time, Systems, vol. 15, no. 3, Kluwer Academic Publishers, 1998, pp. 275 - 298.

Wegener, J.; Baresel, A.; Sthamer, H.: *Evolutionary Test Environment for Automatic Structural Testing*. To appear in the Special Issue of Information and Software Technology devoted to the Application of Metaheuristic Algorithms to Problems in Software Engineering, 2001.

Wegener, J.; Pohlheim, H.; Sthamer, H.: *Testing the Temporal Behavior of Real-Time Tasks using Extended Evolutionary Algorithms*. Proceedings of the 7th European Conference on Software Testing, Analysis and Review (EuroSTAR '1999), Barcelona, Spain, November 1999.