Using Evolutionary Testing to improve Efficiency and Quality in Software Testing

Harmen Sthamer, Joachim Wegener and Andre Baresel

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

Abstract

The development of embedded systems is an essential industrial activity. More than 90% of all electronic components manufactured are used in embedded systems, e.g. in aerospace technology; railway and motor vehicle technology; process and automation technology; communication technology; process and power engineering, as well as in defense electronics. Embedded systems are also used regularly in safety relevant applications. Therefore, the occurrence of errors may endanger human lives or cause costly recalls, for example in the automotive industry. Accordingly, the development of embedded systems must comply with the highest quality requirements and standards.

Analytical quality assurance is of central importance to achieving high quality development of embedded systems. In practice, the most important analytical quality assurance measure is dynamic testing. The thorough testing of developed systems is therefore essential for product quality. The aim of testing is to detect errors in the system under test and to convey confidence in the correct functioning of the system if no errors are found during comprehensive testing.

The effectiveness and efficiency of the test process can be clearly improved by evolutionary testing. This has been successfully proved in several case studies and industrial applications. Evolutionary Tests thus contribute to quality improvement as well as to the reduction of development costs.

1 Introduction

The testing of embedded systems is considerably more complex than the testing of conventional software systems. This is due to the technical features of embedded systems, and to special requirements made on these kinds of systems, e.g. embedded systems usually have to fulfil functional as well as non-functional requirements such as temporal requirements, the computational accuracy of the target system, memory space requirements during program execution, or the synchronization of parallel processes.

Tests are the only procedure that allow dynamical system behavior to be tested in a real application environment, and therefore are the most important quality assurance measure for embedded systems. Interaction with the real application environment is equally important to system reliability. This includes, for example, the employed target hardware, the employed operating system, and the employed compiler. Therefore, testing typically takes up more than 50% of the overall development effort and budget for embedded systems [2].

The most significant weakness of testing is that the postulated functioning of the tested system can, in principle, only be verified for those input situations which were selected as test data. According to Dijkstra [4], testing can only show the existence but not the non-existence of errors. Proof of correctness can only be produced by a complete test, i.e. a test with all possible input values, input value sequences, and input value combinations under all practically possible constraints. In practice, complete testing is usually impossible because of the vast amount of possible input situations. Testing can therefore only be a sampling method. Accordingly, the selection of an appropriate sample containing the most error-sensitive test data is essential to testing. If test data relevant to the practical deployment of the system are omitted, the probability of detecting errors within the software declines. Of all the testing activities – test case design, test execution, monitoring, test evaluation, test planning, test organization, and test documentation – essential importance is thus attributed to test case design [24].

Systematic test case design is indispensable to good test quality because it defines the type and scope of the test. For most test objectives, test case design is difficult to automate:

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

Therefore, test cases have to be defined manually which affects the efficiency and effectiveness of the executed test.

In order to increase the effectiveness and efficiency of the test and thus reduce the overall development costs for embedded systems, a test is required 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 [11] and TPT [10], evolutionary testing [20] is a promising approach for the complete automation of test case design for the remaining three aspects mentioned above.

The Evolutionary Test can be applied to tests of the temporal behavior of systems (e.g. [20] and [25]); it can be used to generate test cases for structural testing (e.g. [1] and [21]), and it enables the automation of safety testing (e.g. [19]). The aim of this work is to increase the efficiency and quality of the tests and to achieve substantial cost savings in system development by a high degree of automation.

In order to carry out an evolutionary test, the test case design has to be transformed into an optimization problem which in turn is solved with meta-heuristic search techniques, such as evolutionary algorithms and simulated annealing. The input domain of the system under test represents the search space in which test data fulfilling the test objectives under consideration are searched for. The Evolutionary Test is generally applicable because it adapts itself to the system under test.

The second chapter introduces the basic principles of evolutionary algorithms and their application to testing; the Evolutionary Test. The third chapter discusses the use of temporal behavior and structural evolutionary testing. The paper concludes with a summary of the most important results.

2 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 characterized by an iterative procedure and work parallel on a number of potential solutions for a population of individuals. Permissible solution values for the variables of the optimization problem are encoded in each individual.

The fundamental concept behind 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 tries to achieve the optimum solution by means of the random exchange of information between increasingly fit samples (recombination), and the introduction of a probability of independent random change (mutation). The adaptation of the evolutionary algorithm is achieved using selection and reinsertion procedures based on fitness. Selection procedures control which individuals are selected for reproduction, depending on the individuals' fitness values. The reinsertion strategy determines how many, and which, individuals are taken from the parent and 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 may depend critically on the definition of a fitness 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 optimization problem.

Fig. 1 provides an overview of a typical evolutionary algorithm procedure. First, a population of guesses as to the solution of a problem is initialized, usually at random. Each individual within the population is evaluated by calculating its fitness. This usually results in a spectrum 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 such a way as to produce a new guess analogous to biological reproduction.

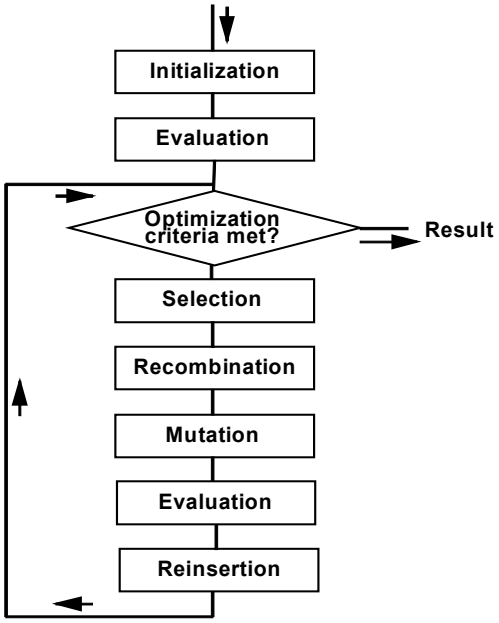


Figure 1: Evolutionary Algorithms

The algorithm combinations are many and varied. Mutation is also 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.

2.1 Application to Software Testing

In order to automate software tests using evolutionary algorithms, the test aim must itself be transformed into an optimization task. 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. Depending on which test aim is pursued, different fitness functions emerge for test data evaluation. If an appropriate fitness function can be defined for the test aim, and evolutionary computation is applied as the search technique, then the Evolutionary Test proceeds as follows.

The initial set of test data is generated, usually at random. In principle, if the test data has been obtained by a previous systematic test, this could also be used as an initial population[22]. The Evolutionary Test could thus benefit from the tester's knowledge of the system under test.

Each individual within the population represents a test datum with which the system under test is executed. For each test datum the execution is monitored and the fitness value determined for the corresponding individual.

Next, test data with high fitness values are selected with a higher probability than those with a lower value and are subjected to combination and mutation processes to generate new offspring test data. It is important to ensure that the test data generated are in the input domain of the test object. The main idea behind evolutionary testing is the combination of interesting test data in order to generate offspring test data that truly fulfil the test objectives. The offspring test data are evaluated by executing the system under test. A new population of test data is formed by merging offspring and parent individuals according to the survival procedures laid down. From here on, the process repeats itself, starting with selection until the test objective is fulfilled or another given stopping condition is reached (compare Fig. 2).

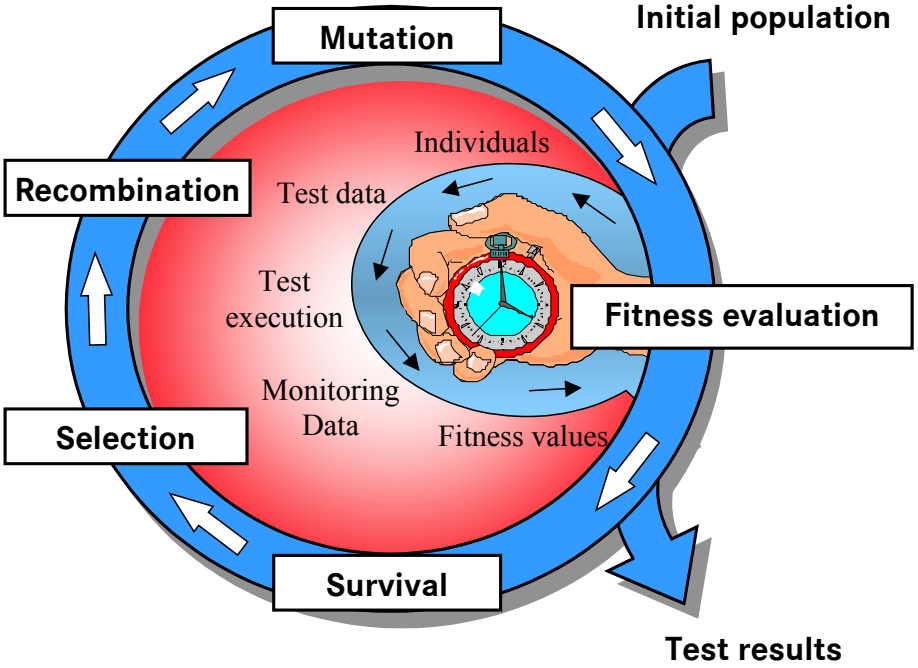


Figure 2: Evolutionary Test

Due to the non-linearity of software (if-statements, loops, etc.), the conversion of test problems into optimization tasks usually results in complex, discontinuous, and non-linear search spaces. Neighborhood search methods such as hill climbing are not suitable in such cases. Therefore, meta-heuristic search methods, such as evolutionary algorithms, are employed because their robustness and suitability for the solution of different test tasks has already been proven in previous work, e.g. [8], [17], and [18]. The suitability of evolutionary algorithms for testing is based on their ability to produce effective solutions for complex and poorly understood search spaces with many dimensions. The dimensions of the search spaces are directly related to the number of input parameters of the system under test. The execution of different program paths, and the nested structures in software systems, lead to multi-model search spaces when testing. Apart from many local optima, these are also marked by jumps and levels of identical fitness values. Input parameter dependencies within the system under test may result in definition gaps. A noisy fitness function may also be caused by internal system states, in this case identical input values may result in different fitness values.

During optimization, evolutionary algorithms identify the *building blocks* of an ideal solution, and store those blocks in the individuals within the population. Building blocks are the partial solutions (genetic modules) of which good solutions are composed. The combination of individuals and different building blocks results in more qualified individuals as optimization continues. As well as being particularly well-suited to the treatment of complex search spaces, evolutionary algorithms also represent a very robust optimization procedure.

There are few prerequisites for the application of the Evolutionary Test. An interface specification of the system under test is required to guarantee the generation of valid input values. For structural testing, the source code of the test object is required. The most important prerequisite is a numeric presentation of the test aim, from which a suitable fitness function for the evaluation of generated test data can be derived. Different fitness functions emerge for test data evaluation

depending on which test aim is pursued. In order to carry out temporal behavior testing, the fitness evaluation of the test object is based on the execution time measured. Fitness values for testing safety requirements are derived from pre- and post conditions of modules. The automation of structural test case design uses the control-flow graph executed by a test datum to form the starting point for the fitness evaluation.

3 Demonstrating the Capabilities of Evolutionary Testing

In this chapter experiments are presented which demonstrate the efficiency and the effectiveness of the evolutionary approach for different applications, e.g. evolutionary temporal and structural testing. In addition, the evolutionary testing approach is compared to systematic test case design, random testing and finally to static analyses. Each experiment was performed ten times using evolutionary testing and random testing in order to achieve statistical mean values.

3.1 Evolutionary Algorithm Settings

The configuration of the evolutionary operators was left unchanged throughout the test experiments. Population size was fixed at 300 individuals. Rank-based fitness assignment was used, i.e. the fitness assigned to each individual depends only on its rank position and not on the concrete value of the overall fitness, as would be the case for proportional fitness assignment. A reasonable selective pressure was applied to ensure that diversity of the population was retained and to avoid a rapid convergence towards a local optimum. Stochastic universal sampling was used as a selection method.

Discrete recombination was applied for the recombination of individuals. Integer mutation was employed using different range parameters. An individual's probability of mutating variables is set to be inversely proportional to its number of variables. For each variable, the mutation probability is 1/number_of_variables, i.e. one mutation within one individual. A reinsertion strategy with a generation gap of 90% was applied in our experiments, i.e. 90% offspring and 10% parents form the next generation. The best parents from the previous generation survive.

An extended population model using several subpopulations was applied to our evolutionary algorithms. The subpopulations utilize different evolutionary algorithms in order to apply a different search strategy, e.g. local searches or global searches. In order to combine local and global search strategies, the best individuals migrate between the subpopulations at regular intervals. The subpopulations also compete with each other. Strong subpopulations receive more individuals whilst other subpopulations diminish in size. This results in an automatic distribution of resources. The evolutionary test was concluded as soon as complete branch coverage had been achieved or if a maximum number of generations was produced. For each partial aim the maximum number of test data to be created was limited to 200 generations.

3.2 Testing Temporal Behavior of Real-Time-Systems

Most embedded systems are subject to temporal requirements. This is due to reasons of operational comfort, e.g. short system reaction times to user commands, or due to the requirements of technical processes that are controlled by the system. Therefore, embedded systems have to be thoroughly tested not only with regard to their functional behavior, but also to detect existing deficiencies in temporal behavior. Existing test methods are unsuitable for the examination of temporal correctness.

Even for an experienced tester, it is virtually impossible to find such inputs by analyzing and testing the temporal behavior of complex systems manually. Effects of modern processor architectures with pipelining, data caching and instruction caching as well as the use of system calls, parallelism, and optimizing compilers on the timing behavior of a system can hardly be assessed by the tester.

When testing the temporal behavior 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 individuals generated 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 objective of the test and combined in order to obtain test data with even longer or shorter execution times. The test is terminated if an error in temporal behavior is detected or a specified termination criterion is reached. If a violation of the system's predetermined temporal limits is detected, the test was successful and the system has to be corrected. Evolutionary testing enables a fully automated search for extreme execution times to be carried out.

Besides the evolutionary test a static analysis was also carried out for the test objects. In order to carry out the static analysis, the target processor and the memory available must be replicated in a simulation. The minimum requirement for the static analysis is that the upper and lower limits for loops must be specified.

The static analysis run time prediction represents a safe and guaranteed yet pessimistic (and not the closest predictable) upper limit for the run time of the test object, since the estimate is only based on the control flow graph (CFG). Here guaranteed means that, for example, no longer execution times exist when focusing on the worst case execution time (WCET). However, the path belonging to this predicted execution time is generally infeasible. In order to carry out more exact analyses and thus achieve a closer prediction of the WCET and best case execution time (BCET), a data flow analysis must also be performed in order to exclude those infeasible paths which would lead to a pessimistic prediction.

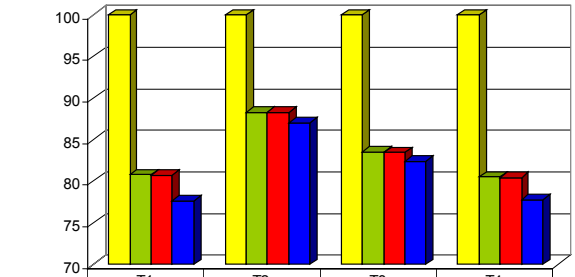
For this reason, the program execution is simulated manually, beginning with the start of the source text. If one reaches a point in the CFG of the program, which is not executable due to a logical correlation with a place higher up in the program, since there is no input set which makes this path possible, this correlation is added in the form of a functional constraint.

This step is an iterative process which must be repeated until the CFG has been completely simulated and no contradiction has been found. One can then be sure that the worst case path found is feasible and the WCET of the path represents the closest upper limit ($SA_{\_feas}$) amongst the given constraints. This is very costly, as it is generally performed manually.

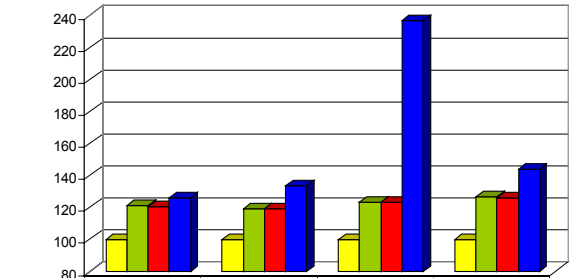
3.2.1 Experimental results

Previous work has already shown that evolutionary testing has always achieved better results than random testing (e.g. [23] and [26]). 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, see Figure 3 and [13]. Figure 3 displays a comparison of

WCET and BCET of four different test objects (T1 to T4 from the field of motive power engineering. The execution times, found using static analysis (SA), evolutionary testing (ET) and random testing (RT) respectively, are presented. A 167 CPU with a 20 MHz clock speed served as the target processor. The execution times, using dynamic methods, were determined using hardware timers from the target environment with a resolution of 200ns. The same amount of test cases were generated for the random test as for the evolutionary test. The evolutionary and random test are implemented in the test system TESSY, [24] and [27], developed by DaimlerChrysler Research.



	T1	T2	T3	T4
SA_long	3118	802	5702	4088
SA_feas	2520	708	4764	3292
ET	2516	708	4760	3288
RT	2420	698	4698	3178



	T1	T2	T3	T4
SA_short	1594	414	1904	1936
SA_feas	1928	492	2344	2448
ET	1924	492	2344	2436
RT	2008	552	4512	2784

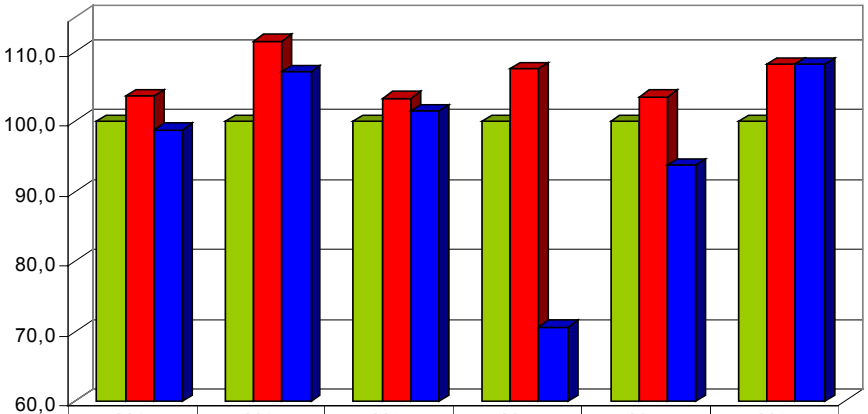
Figure 3: Comparison of static, evolutionary and random execution times found

Figure 3 displays the results of four tasks investigated for WECT on the left hand side and BECT on the right hand side. The numbers displayed are the execution times in processor. The diagrams themselves are normalized and the static analyses' results $SA_{\_long}$ and correspondingly $SA_{\_short}$ are set to 100%, since these are extreme execution times which are guaranteed and cannot be exceeded.

The static analysis calculates an absolute longest $SA_{\_long}$ and a shortest prediction $SA_{\_short}$ of the execution time for a task, which normally constitute an infeasible path, not dynamically executable, and therefore not accurate. In order to achieve an accurate static analysis, these infeasible paths have to be excluded which normally involves an enormous amount of manual effort. In contrast, $SA_{\_feas}$ is the longest or shortest feasible (dynamic) execution time, reached by excluding iteratively manually infeasible paths until a feasible path has been identified. The WCET results show that the feasible longest execution time possible deviates from the purely static longest prediction ($SA_{\_long}$) by between 12% and 20%. The results are similar for the BCET. Here the shortest feasible time is longer than that of the pure static analysis ($SA_{\_short}$) by between 20% to 25%. As one can see, the ET produced nearly the same results as the static analyses ($SA_{\_feas}$) when the infeasible paths are excluded. These small differences can have a blurring effect because of the timer's resolution. The only difference is that ET is completely automatable and that dynamic and environmental aspects are taken into account. Accurate results can be achieved within minutes. An accurate static analysis prediction is very time consuming, because, for every new software task, its extremely time consuming infeasible paths have to be excluded; this is normally done manually. The RT result gives an impression of the effectiveness using ET.

The Evolutionary Test has also attained convincing results as compared to the systematic developer tests, as Figure 4 illustrates. The six tasks (M1 to M6) are derived from an engine control system for six- and eight-cylinder blocks which contain several tasks that have to fulfill 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 behavior, defined by the developers, are based on the functional specification of the system as well as on the internal structures of the tasks. The developer tests achieved full branch coverage for each

task. Evolutionary testing was used to verify these results. The tests were performed on the target processor later used in the vehicles.



	M1	M2	M3	M4	M5	M6
DT	1344	2168	2168	1280	1152	1080
ET	1392	2416	2240	1376	1192	1168
RT	1328	2320	2200	904	1080	1168

Figure 4: Results for the engine control system tasks

Figure 4 shows the longest execution times determined by the developers with systematic testing (DT) in comparison with the results achieved by evolutionary testing (ET) and random testing (RT). The results of the tests carried out by the developer are set to be 100 %. The execution times are measured in processor cycles. The size of the tasks varied from 39 lines of code to 119; the number of input parameters from 9 to 32.

A comparison of the results shows that evolutionary testing found the longest execution times for all the given tasks among these three testing methods. The tests carried out by the developer did not attain the longest execution time. In three cases, the developer's test results are even worse than those of the random test. For the other three tasks the results are better than those of the random test. The latter only finds the longest execution time for task M6. The longest execution time found by the random test in task M4 lies more than 35 % below the value determined by the Evolutionary Test, and 30 % below that of the developer's tests.

The excellent performance of the Evolutionary Test in comparison to the developer tests also shows the effectiveness of the Evolutionary Test in comparison to function-oriented and structure-oriented testing methods. The results are particularly astonishing because evolutionary testing treats the software as black boxes whereas the developers are familiar with the function and structure of their system. This could be explained by the use of system calls, the effects of which on the temporal behavior are difficult to rate for developers.

3.3 Evolutionary Structural Testing

Structural testing is widespread in industrial practice and stipulated in many software-development standards. Statement, branch, and condition testing are common examples. 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 selected structural test criterion.

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

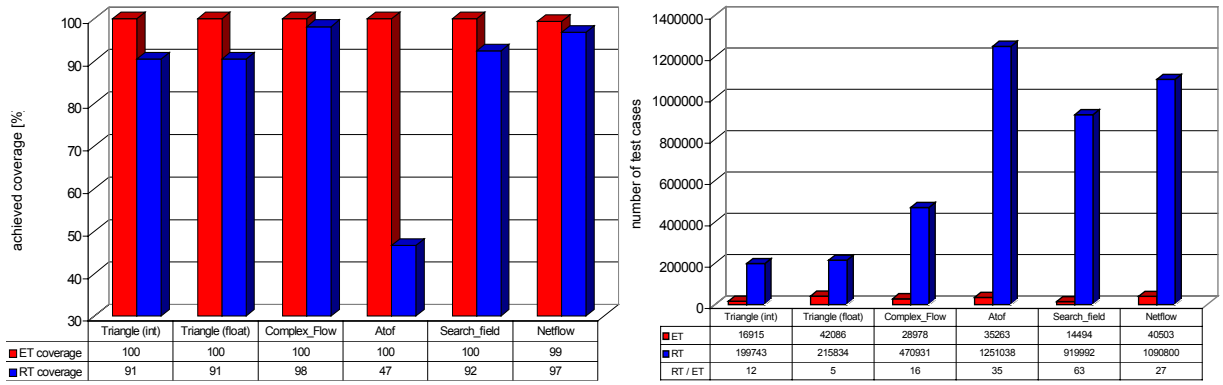
partial aim an individual fitness function is formulated and a separate optimization is performed to search for a test datum executing the partial aim. The set of test data found for the partial aims then serves as the test data set for the coverage of the structure test criterion. In order to direct the search toward program structures not covered, the fitness function computes a distance for each individual that indicates how far away it is from executing the desired program structure. Individuals closer to the execution of the desired program structure are selected as parents and combined to produce offspring individuals.

The fitness functions of the partial aims consist of two components – the *approximation level* and the *local distance* calculation. The approximation level supplies a figure for an individual that gives the number of branching nodes lying between program structures covered by the individual and the desired program structure. For this computation, only branching nodes that contain an outgoing edge resulting in a miss of the desired program structure are taken into account. In addition, the calculation of the local distance is made in order to distinguish between different individuals executing the same program branch. A distance to the execution of the sibling branch is calculated for the individual by means of the branching conditions in the branching node in which the target node is missed.

Although only one partial aim after the other is processed by the Evolutionary Test, the execution of a test datum usually leads to passing several partial aims. Thus, the test soon focuses on those program structures which are difficult to reach. After the processing of all partial aims, the tester is provided with a minimal amount of test data, leading to an execution of all reached partial aims. A detailed definition of the fitness functions and the test environment consisting of a parser, an instrumenter, a test driver generator, and a test control for the automation of the evolutionary structural test can be found in [21] and [1].

3.3.1 Experimental 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. A complete coverage was achieved for most test objects. Figure 5 shows the results obtained from the examination of test objects from different application fields.



	Triangle (int)	Triangle (float)	Complex_Flow	Atof	Search_field	Netflow
ET coverage	100	100	100	100	100	99
RT coverage	91	91	98	47	92	97

	Triangle (int)	Triangle (float)	Complex_Flow	Atof	Search_field	Netflow
ET	16915	42086	28978	35263	14494	40503
RT	199743	215834	470931	1251038	919992	1090800
RT / ET	12	5	16	35	63	27

Figure 5: Structural test results for various complex tasks

On the left hand side the results of the mean coverage achieved by evolutionary and random testing are displayed. On the right hand side of Figure 5, the mean number of test data generated by the evolutionary, the random tests and its ratio are presented. One branch in the Netflow() function is infeasible. This leads to the highest possible coverage of 99.3%. Evolutionary testing performed notably better than random testing for all the functions mentioned.Even though

between 5 and 63 times more test data were generated for random testing, the coverage reached is not as good as for evolutionary testing.

4 Conclusion

The overall results show evolutionary testing to be a promising approach for fully automating test case design for various testing methods and aims. In order to increase the effectiveness and efficiency of the test, and thus to reduce the overall development costs for software-based systems, we require a test which is systematic and extensively automatable.

Evolutionary testing is based on the idea of searching for relevant test cases in the input domain of the system under test with the help of evolutionary algorithms. Evolutionary testing enables the complete automation of test case design whenever the test aim can be expressed numerically, e.g. when performing temporal behavior testing or structural testing.

Due to the full automation of evolutionary testing, the effectiveness and efficiency of the test process can be clearly improved in all these application fields. The system under investigation can be tested with a large number of different input situations both for testing temporal behavior and for safety tests. In most cases, more than several thousand test data are generated and executed within a few minutes. Evolutionary tests thus contribute to quality improvement as well as to the reduction of development costs for embedded systems.

The test generation results provided by evolutionary testing for temporal behavior are efficient and of high quality compared to the results achieved with static analyses. Whilst the Evolutionary Test is fully automatable, the static analysis involves extensive manual work in order to achieve an exact estimate, e.g. the exclusion of infeasible paths. Additional time-consuming work for the static analysis, e.g. the simulation of the machine model of the CPU, memory, the consideration of cache and pipelining effects makes it very expensive to provide this method for new CPUs.

The application scope of Evolutionary Tests goes further than the work described in this paper. Additional application fields are, for instance, functional [9] and robustness tests [16].

We are also investigating the application of evolutionary structural tests to testing temporal behavior of systems. The idea is to pre-determine program paths, identified as worst-case execution time paths by means of static analyses (e.g. [12] and [15]), as test aims for the evolutionary structural test. If a test datum can be found that executes the path we can be sure that this is the longest execution time which it is possible to obtain. As a result of 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

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

[2] Beizer, B.: *Black-Box Testing – Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, 1995.

[3] Conrad, M., Dörr, H., Fey, I., Yap, A.: Model-based Generation and Structured Representation of Test Scenarios, Proceedings of the Workshop on Software-Embedded Systems Testing, Gaithersburg, Maryland, USA, 1999.

[4] Dijkstra, E. W., Dahl, O. J., Hoare, C. A. R.: 'Structured programming', Academic Press., 1972.

[5] Grochtmann, M., Wegener, J.: Evolutionary Testing of Temporal Correctness. Proceedings of the 2nd International Software Quality, Week Europe (QWE' 1998), Brussels, Belgium, November 1998.

[6] Gross, H.-G., Jones, B. und Eyres, D.: Structural performance measure of evolutionary testing applied to worst-case timing of real-time systems, IEE Proc.-Softw., Vol. 147, No. 2, pp. 25 – 30, April 2000.

[7] 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.

[8] Jones, B.-F., Sthamer: H.-H., Eyres, D.: Automatic structural testing using genetic algorithms. Software Engineering Journal, vol. 11, no. 5, pp. 299 – 306, 1996.

[9] Jones, B.F., Sthamer, H.- -H., Yang, X., Eyres, D.E.: The Automatic Generation of Software Test Data Sets using Adaptive Search Techniques. Proceedings of Software Quality Management '95, Seville, Spain, pp. 435 - 444, 1995.

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

[11] 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.

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

[13] 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.

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

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

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

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

[18] 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.

[19] Tracey, N., Clark, J. and Mander, K.: Automated Program Flaw Finding using Simulated Annealing. Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '98), Clearwater Beach, Florida, USA, pp. 73 – 81, 1998.

[20] 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, pp. 275 - 298, 1998.

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

[22] Wegener, J., Grimm, K., Grochtmann, M., Sthamer, H. und Jones, B.: Systematic Testing of Real-Time Systems. Proceedings of the Fourth European International Conference on Software Testing, Analysis & Review, Amsterdam, Netherlands, 1996.

[23] Wegener, J.; Grochtmann, M.; Jones, B.: Testing Temporal Correctness of Real-Time Systems by Means of Genetic Algorithms, Proceedings of the 10th International Software Quality Week (QW '97), San Francisco, USA, May 1997.

[24] Wegener, J. and Pitschinetz, R.: *TESSY – Yet Another Computer-Aided Software Testing Tool?* Proceedings of the Second International Conference on Software Testing, Analysis and Review, Bruxelles, Belgium, 1994.

[25] 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.

[26] Overview of Functional and Evolutionary Testing: www.systematic-testing.com

[27] The Test system Tessy: www.razorcat.de