Testing the Temporal Behavior of Real-Time Software Modules using Extended Evolutionary Algorithms

Hartmut Pohlheim, Joachim Wegener,
DaimlerChrysler, Research and Technology, Alt-Moabit 96a, 10559 Berlin, Germany
phone: +49 30 39982 {456, 232}, fax: 107
hartmut.pohlheim@daimlerchrysler.com

Many industrial products are based on the use of embedded computer systems. Usually, these systems have to fulfill real–time requirements, and correct system functionality depends on their logical correctness as well as on their temporal correctness. Therefore, the developed systems have to be thoroughly tested in order to detect existing deficiencies in temporal behavior, as well as to strengthen the confidence in temporal correctness.
Existing test methods are not specialized in the examination of temporal correctness. For this reason, new test methods are required which concentrate on determining whether the system violates its specified timing constraints. Normally, a violation means that outputs are produced to early, or their computation takes to long. The task of the tester therefore is to find the input situations with the longest or shortest execution times, in order to check whether they produce a temporal error. It is virtually impossible to find such inputs by analyzing and testing the temporal behavior of complex systems manually. However, if the search for such inputs is interpreted as a problem of optimization, evolutionary computation can be used to find the inputs with the longest or shortest execution times. This search for accurate test data by means of evolutionary computation is called evolutionary testing.
Experiments using evolutionary testing on a number of real world programs have successfully identified new longer and shorter execution times than had been found using other testing techniques. Evolutionary testing, therefore, seems to be a promising approach for the verification of timing constraints.
For the optimization we employed extended Evolutionary Algorithms. This includes the use of multiple subpopulation, each using a different search strategy. Additionally, competition for limited resources between these subpopulation have taken place, providing an efficient distribution of the resources.

1 Testing Real-Time Systems

In real–time computing the correctness of the system does not only depend on the logical result of the computation, but also on the time when the results are produced. Real–time systems play an important role in our life, and they cover a spectrum from the very simple to the very complex. Examples of current real–time systems include applications from the field of aerospace, automotive, and railway electronics. These are often safety–relevant systems which, in case of error, can cause considerable material damage or can even endanger human lives. At present, there is still a lack of suitable methods and tools for the development of real–time systems which permit a coherent treatment of correctness, timeliness, and fault tolerance in large scale distributed computations. For this reason, real–time system design is mostly done unsystematically, and timing constraints are verified with ad hoc techniques, or with costly and extensive simulations (cf. [19]).

Testing is one of the most complex and time-consuming activities within the development of real-time systems [7]. It typically consumes 50 % of the overall development effort and budget since embedded systems are much more difficult to test than conventional software systems ([3], [9], [2], [22]). The examination of additional requirements like timeliness, simultaneity, and predictability make the test costly. In addition, testing is complicated by technical characteristics like the development in host-target environments, the strong connection with the system environment, the frequent use of parallelism, distribution and fault-tolerance mechanisms, as well as the utilization of simulators.

Nevertheless, systematic testing is an inevitable part of the verification and validation process for software-based systems. Testing is the only method that allows a thorough examination of the test object's run-time behavior in the actual application environment. Dynamic aspects like the duration of computations, the memory actually needed during program execution, or the synchronization of parallel processes are especially important for the correct functioning of real-time systems.

Testing is aimed at finding errors in the systems and giving confidence in their correct behavior by executing the test object with selected inputs. For testing real–time systems, the examination of the logical system behavior alone is not sufficient. Additionally, real-time systems must be tested for compliance with their timing constraints.

An investigation of existing software test methods shows that a number of proven functional and structural test methods are available for examining logical correctness. For functional tests, test cases are derived from the requirements specification. A functional test procedure, for example, is the classification-tree method [4], which has already been used successfully for the functional test of different real-time systems from various application fields [5]. When using structure-oriented test methods, test cases are determined on the basis of the program code. The aim of the test is to obtain a high coverage according to the selected test criterion, e.g. statement or branch coverage. An important principal disadvantage of structural testing

Pohlheim, H. and Wegener, J.: Testing the Temporal Behavior of Real-Time Software Modules using Extended Evolutionary Algorithms.
full paper of the published poster paper: Banzhaf, W. (ed.): GECCO'99 - Proceedings of the Genetic and Evolutionary Computation Conference, San Francisco, CA: Morgan Kaufmann, p. 1795, 1999.

is that the tester cannot check whether all specified requirements have been implemented into the system. Additionally, the tester must take into account that an instrumentation of the test object to monitor program execution may change the run-time characteristics of the program. Probe effects, i.e. deviations from the real system behavior are possible. Furthermore, the program under test may no longer fit into the target machine if the code is expanded to monitor execution [8].

For examining temporal correctness there are no specialized test methods available which are suited for industrial use [23]. For this reason, we have developed and examined a new approach for testing temporal behavior which is based on the use of evolutionary algorithms, namely evolutionary testing. Our work investigates the effectiveness of evolutionary algorithms to validate the temporal correctness of embedded systems by establishing the maximum and minimum execution times. Good results have been achieved in our experiments.

Section 2 contains an overview of evolutionary testing and describes how it is applied to examine the temporal behavior of real-time systems. The results of several experiments will be described and discussed in Section 3. Section 4 gives concluding remarks and a short outlook on current and future work.

2 Evolutionary Testing

The major objective of testing is to find errors. Real-time systems are tested for logical correctness by standard testing techniques such as the classification-tree method [5]. A common definition of a real-time system is that it must deliver the result within a specified time interval and this adds an extra dimension to the validation of such systems, namely that their temporal correctness must be checked.

The temporal behavior of real-time systems is defective when input situations exist in such a manner that their computation violates the specified timing constraints. In general, this means that outputs are produced too early or their computation takes too long. The task of the tester therefore is to find the input situations with the shortest or longest execution times to check whether they produce a temporal error.

The search for the shortest and longest execution times can be regarded as an optimization problem to which Evolutionary Algorithms seem an appropriate solution.

2.1 Investigation of search space

In contrast to other optimization methods, evolutionary algorithms are particularly suited for problems involving large numbers of variables and complex input domains. Even for non-linear and poorly understood search spaces evolutionary algorithms have been used successfully [23].

With the exception of very simple real-time systems, the temporal behavior always forms a very complex multi-dimensional search space with many plateaus and discontinuities, because the execution times for several input data executing the same program path are identical; whereas the execution of different program paths leads to irregular changes of the execution times. Two examples are provided in figure 1.

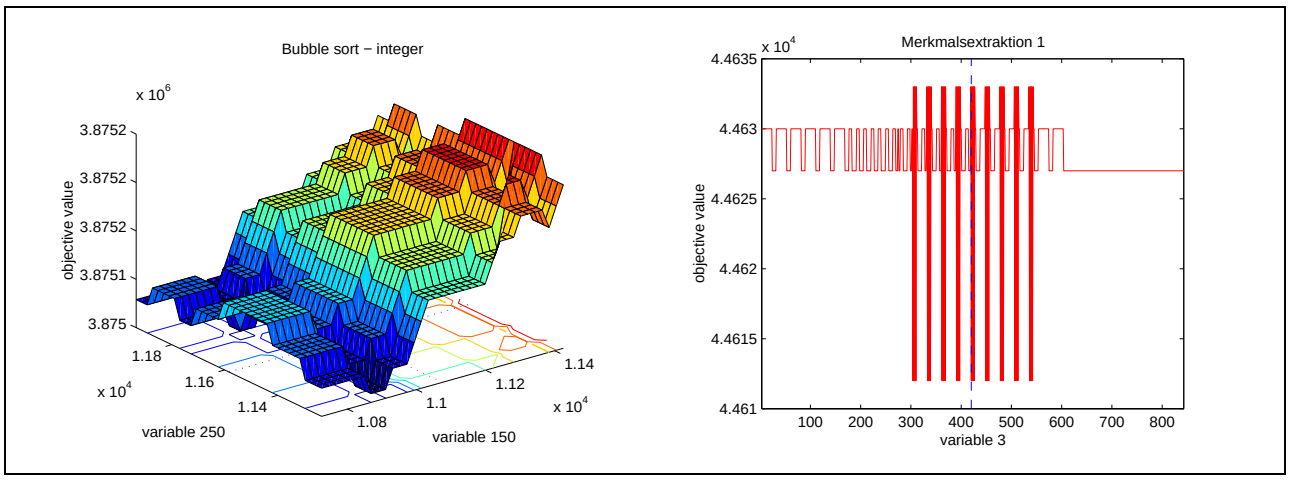

Figure 1: Visualization of execution times for a very small area of the search space of two software systems; left: Bubblesort algorithm - variation of two variables (out of 500 variables), right: feature extraction - variation of just one variable (out of more than 800 variables)

2.2 Application of Evolutionary Algorithms

When using evolutionary testing for determining the shortest and longest execution times of test objects, each individual of the population represents a test datum with which the system under test is executed. In our experiments the initial population is generated at random. In principle, if test data has been obtained by a systematic test, these could also be used as an initial population. Thus, the evolutionary approach benefits from the tester's knowledge of the system under test.

For every test datum, the execution time is measured. Afterwards, the population is sorted according to the execution times determined. The fitness assigned to each individual depends only on its position in the individuals rank and not on the actual execution time. This rank-based fitness assignment overcomes some problems which apply to the proportional fitness assignment, and behaves in a more robust manner than the proportional fitness assignment [24]. In this way, the execution time for each test datum determines its fitness value. If one searches for the worst-case execution time, test data with long execution times obtain high fitness values. Conversely, when searching for the best-case execution time, individuals with short execution times obtain high fitness values.

Members of the population are selected with regard to their fitness and subjected to recombination and mutation to generate new test data. By means of selection, it is decided which test data are chosen for reproduction. In order to retain the diversity of the population, and to avoid a rapid convergence towards a local optimum, a moderate selective pressure was applied. In our experiments, we used stochastic universal sampling [1] as selection method.

For the recombination of test data, two recombination methods are applied:

- discrete recombination [14] and
- multi-point recombination (an extension of two-point crossover).

For mutation we also employed multiple methods:

- integer mutation using different range and precision parameters (an integer version of the mutation operator of [14]) and
- swap mutation using different ranges (exchange/swap of variables inside one individual, the range defines how far away from each other the variables are to be exchanged).

The probability of mutating variables of an individual is set to be inversely proportional to its number of variables. The more dimensions one individual has, the smaller the mutation probability for each single variable. Correspondingly, for each variable, the mutation probability is 1/number_of_variables.

Afterwards it has to be checked if the generated test data are in the input domain of the test object. Invalid data are readjusted. The individuals produced are then evaluated by executing the test object with them and measuring the execution times.

The new individuals are united with the previous generation to form a new population according to the reinsertion procedures laid down. We apply a reinsertion strategy with a generation gap of 90% (that means, just 90% offspring are produced). The next generation therefore contains 90% offspring and 10% parents. Thus, the best parents from the previous generation survive.

Figure 1 shows the structure of the evolutionary algorithm employed.

We also introduce an extended population model inside our evolutionary algorithms. Besides using multiple subpopulation (regional population model or migration model) and migration, every subpopulation employed different evolutionary parameters (multiple strategies). Competition between the subpopulation takes place every couple of generations. Successful subpopulation receive more individuals, other subpopulation shrink in size. Thus, an automatic distribution of resources takes place.

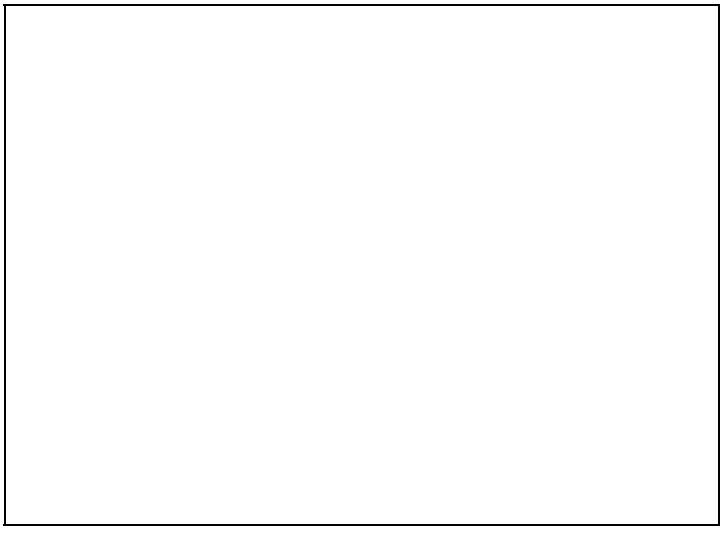

Figure 1: Structure of the extended Evolutionary Algorithm used

For a detailed discussion of the mentioned Evolutionary Algorithms and the introduced extensions see [16]. The Genetic and Evolutionary Algorithm Toolbox for use with Matlab - GEATbx [15] contains an implementation of these methods and was used for the experiments.

The evolutionary process repeats itself, starting with selection, until a given stopping condition is reached, e.g. a certain number of generations is reached, or an execution time is found which is outside the specified timing constraints. In this case, a temporal error is detected and the test is successful. If all the times found meet the timing constraints

specified for the system under test, confidence in the temporal correctness of the system is substantiated. However, in this case it becomes considerably more difficult to decide when the test should be terminated in order to establish sufficient confidence in the temporal correctness of the test object. Therefore, DaimlerChrysler Research and Technology and the Stanford University are currently working on developing suitable stopping criteria [21].

Evolutionary testing enables a totally automated search for extreme execution times. The test especially benefits from the fact that test evaluation concerning temporal behavior is usually trivial. Contrary to logical behavior, the same timing constraints apply to large numbers of input situations.

In the experiments presented in the following section, evolutionary testing was stopped after a predefined number of generations, specified according to the complexity of the test objects with respect to number of input parameters and lines of code.

For a longer description of the used algorithms and methods consider one of the many available publications (cf. [12]).

2.3 Testing temporal system behavior

We have used Evolutionary Algorithms in several experiments to determine the shortest and longest execution times of different systems. For the measurement of the execution times two implementations were used:

1. The performance measurement tool Quantify [17] was used to measure execution times. The experiments described were carried out on a SPARCStation 20 running with 200 MHz under Solaris 2.5. The duration of executions was measured in processor cycles to rule out overheads by the operating system. Thus, the execution times reported were the same for repeated runs with identical parameters.
   Using this method, the example system bubblesort was examined.
2. The execution times were determined using hardware timers of the target system. The hardware timer is started before executing the test datum and returns the time value directly after the execution finishes.
   On the target system we tested software modules from a current motor control project.

In Figure 2 an overview of the integration between Evolutionary Algorithm and the time measurement equipment is given. The scheme is similar for both methods described. In Figure 2 the second variant using a hardware timer of the target system is depicted.

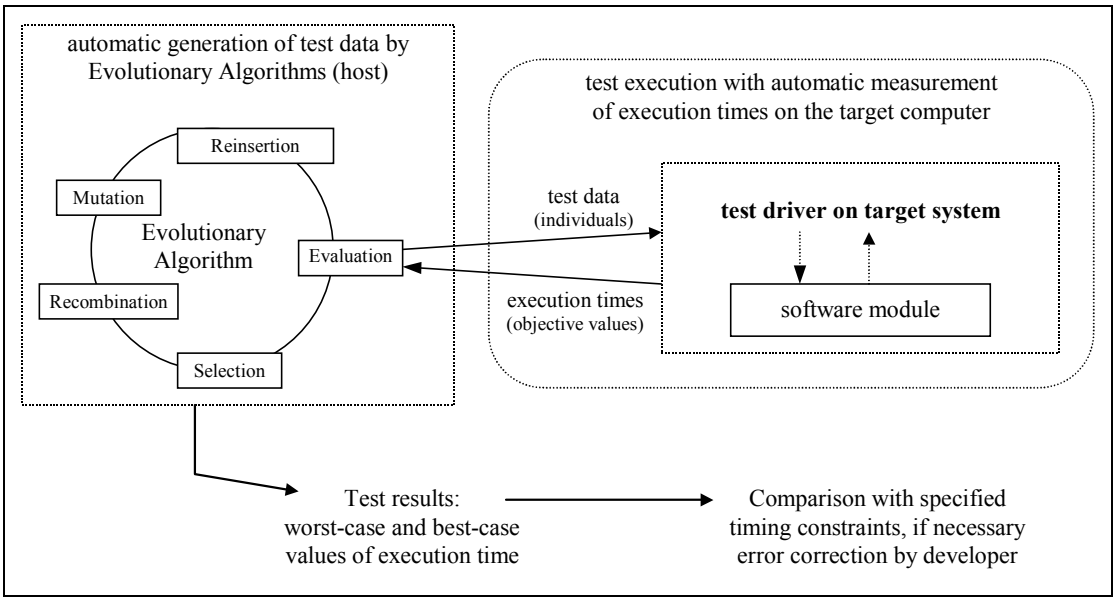

Figure 2: Scheme of evolutionary testing for temporal correctness

3 Application of Evolutionary Testing

Over the last three years many experiments involving evolutionary testing were carried out at DaimlerChrysler, Research and Technology. In this paper we describe results from two application fields:

- The first test object is the bubblesort algorithm with 30 lines of code and 500 parameters. This example system was used for examining the optimization algorithms and for finding appropriate evolutionary parameters. Because of the algorithmic simplicity of the system, the minimum and maximum execution times could be easily calculated (fully sorted list and list sorted in reverse order). The execution times were measured in processor cycles using Quantify.

- The second test object is a motor control system. It contains several modules which have to fulfill timing constraints. These modules were tested on the target processor later used in the vehicles. The execution times were measured by means of hardware timers.

3.1 Test of the bubblesort algorithm

In order to find the longest and shortest execution times, the example system bubblesort may be solved as parameter optimization problem or as on ordering problem. A combination of both methods is possible as well (use of multiple strategies, see section 2.2). Which of the algorithm variants or which combination is best suited for the solution of this problem was derived by the results of different optimization runs using multiple strategies and competition between these strategies.

The following operators were used for the solution:

- 300 individuals (6 subpopulation with 50 individuals each) over 1000 generations (normally we would use more individuals for a problem with 500 variables. However, more individuals would extend the optimization times to more than one day),
- Selection: stochastic universal sampling, generation gap of 0.9,
- Recombination: double point and discrete recombination,
- Mutation: integer mutation and exchange/swap mutation, each with a mutation range of [0.2, 0.02, 0.002] and a mutations precision of 16,
- Migration: 20% every 20 generations,
- Competition (resource distribution) between subpopulation every 5 generations, competition rate of max. 5% of the worst individuals of the subpopulation, subpopulation minimum of 17 individuals.

The recombination operators are equally well suited for this problem. The largest differences are constituted by the mutation operators. The integer mutation changes variables at their actual position inside the individual. Just the value of this variable is more or less changed. By using different mutation ranges the size of these mutation steps can be adjusted. This outlines the kind of search: a large mutation range produces large mutation steps and leads to a rough search, a small mutation range conversely to a fine search. The mutation precision defines the distribution of the mutation steps inside the mutation range.

The exchange or swap mutation exchanges the values of two variables inside one individual. Thus, not the variable values are changed, but the position inside the individual where these values appear. The mutation range predetermines how large the maximum distance between the variables to be exchanged is; similar to the integer mutation. The exchange mutation is a simple mutation operator for ordering problems that seemed appropriate for the bubblesort problem.

The following diagrams show the results of one of the many optimization runs of the bubblesort system. A search for the minimum execution time was carried out.

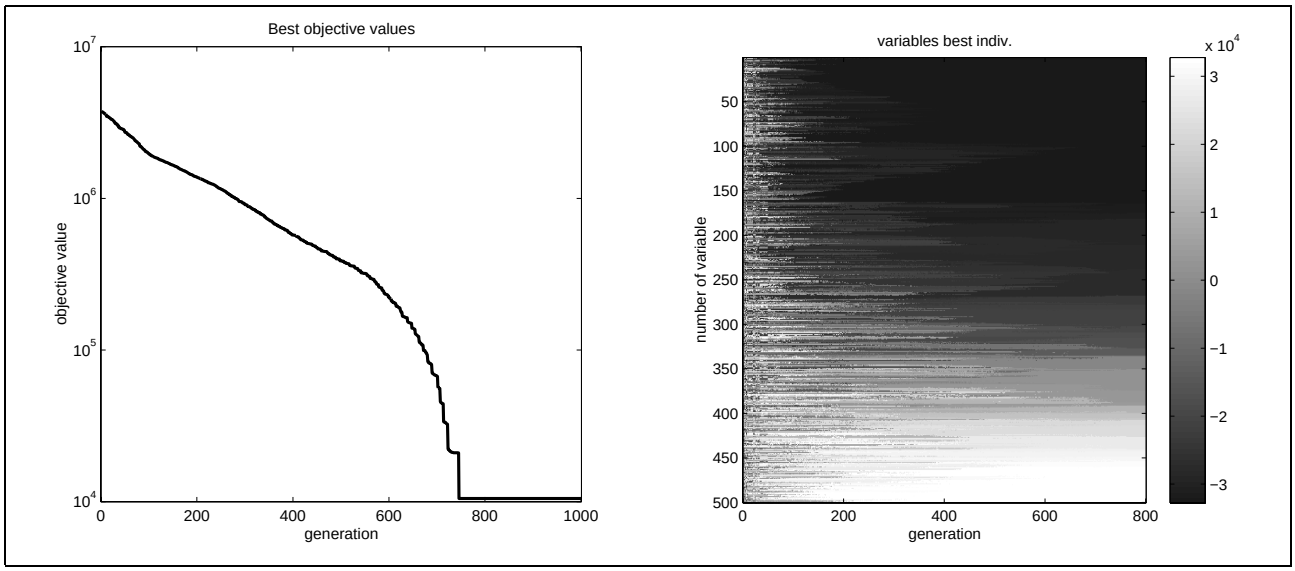

Figure 3: Results of bubblesort optimization; left: objective value of the best individual during all generations (convergence diagram); right: variable values of the best individual over all generations

Figure 3 shows the course of the objective values of the best individuals during all generations (left diagram). The logarithmic scaling shows clearly, that in generation 760 the optimum was reached. A sorted list with 500 entries was derived, controlled only by the execution time. Afterwards, the variables of the best individuals (right diagram) show only small changes which have no effect on the execution time.

In this example we used multiple strategies and competition between these strategies. During the analysis of the results we wanted to answer the question which of the used mutation operators and the accompanying parameters were well-suited for this problem. The analysis uses the two diagrams in figure 4, ranking of the subpopulation (left diagram) and relative size of the subpopulation (right diagram).

When looking at the ranking of the subpopulation (figure 4 left), a low rank characterizes a successful subpopulation (and thus a successful strategy). During the shown run, subpopulation 4 and 6 were equally successful until generation 350. Later on, subpopulation 4 was more successful than all other subpopulation. Beginning with generation 760, subpopulation 6 was more successful, but at this point the run had already reached the optimum. The ranking of the subpopulation corresponds with the size of the subpopulation. Until generation 350, subpopulation 4 and 6 have more or less the same amount of individuals. From then on the size of subpopulation 4 grows up to the maximum value.

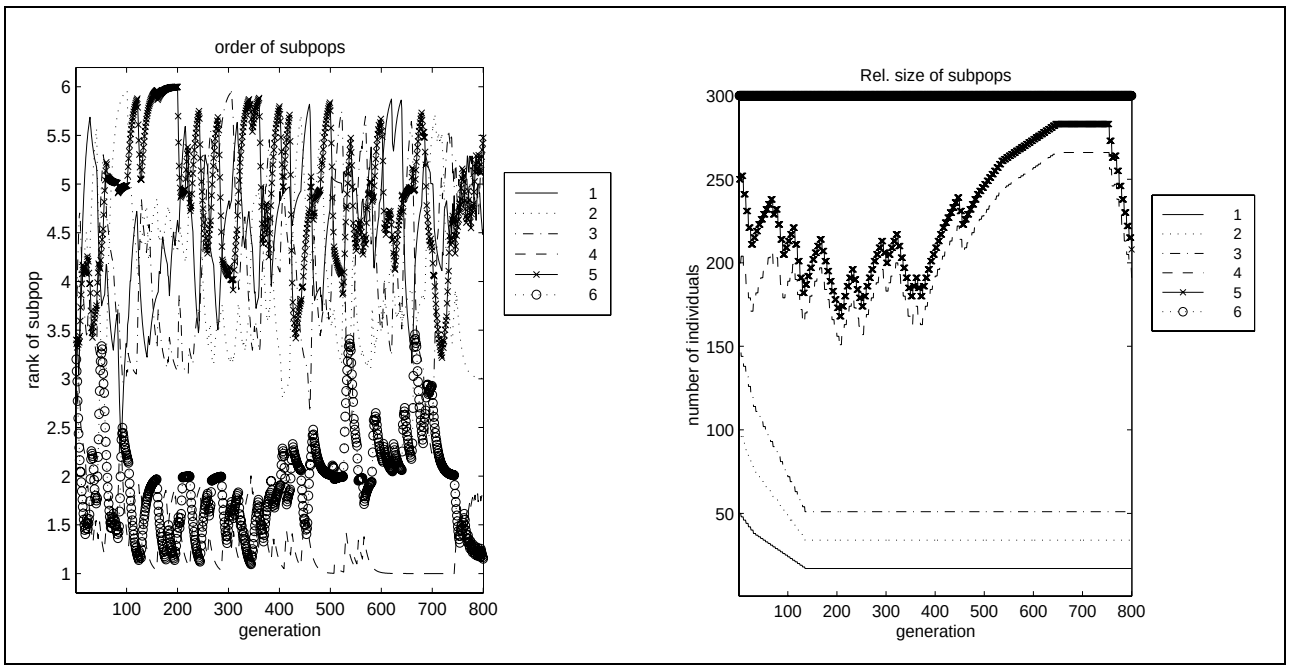

Figure 4: Results of bubblesort optimization, comparison between multiple strategies; left: ranking of subpopulation, right: size of subpopulation

From these results could be derived that the search strategies of subpopulation 4 and 6 are best suited for the solution of the bubblesort system. Subpopulation 4 and 6 use swap mutation with middle and small mutation steps (mutation range of 0.02 and 0.002) respectively.

In a following optimization run we employed this result. We used only two subpopulation with the above strategies (swap mutation with middle and small mutation range). The number of individuals per subpopulation was set to 75, thus only 150 individuals were used. The optimum was found after 1050 generations. However, as we used fewer individuals, the overall number of objective function calls was much lower: 140.000 (1050 generations times 135 individuals per generation) function calls for the second test run compared with 200.000 (760 times 270) objective function calls for the first test run with 6 subpopulation. Accordingly, the overall computation time was much lower (16 hours compared to 24 hours before, 99% of this time were used by the objective function evaluations).

3.2 Test of the motor software modules

The test of the motor software modules could be carried out on the target system. Thus, the time needed for the objective function call was much lower (30-50 seconds per generation, nearly independent of the number of individuals to evaluate). Because of the lower number of module parameters (variables of the Evolutionary Algorithm), an optimization run needed just 50 to 100 generations to reach a good result.

For the application of the results derived from the optimization runs with the bubblesort system, we used multiple subpopulation and employed different strategies and competition between subpopulation. Since the software modules of the motor control system are parameter optimization problems (and not ordering problems) we used parameters best suited for parameter optimization (discrete recombination and integer mutation with multiple ranges). The other parameters are similar to the one used for the solution of the bubblesort system.

The results of some of our optimizations are shown in table 1. For these systems we just report the found maximum execution time, because only the worst-case execution time is of importance for the correct functioning of the motor control system.

For the motor software modules the maximum execution times are not known for sure. Therefore, the results of evolutionary testing were compared to the maximum execution times determined by the developers with systematic testing. These values are shown in table 1 in the third column labeled developer test.

	max. execution time in µs			
module name	evolutionary testing	developer test	lines of code	module parameters
module zr2	69,6 µs	67,2 µs	41	18
module t1	120,8 µs	108,4 µs	119	18
module mc1	112,0 µs	108,4 µs	98	17
module mr1	68,8 µs	64,0 µs	81	32
module k1	59,6 µs	57,6 µs	39	14
module zk1	58,4 µs	54,0 µs	56	9

Table 1: Maximum execution times of motor control software modules determined by evolutionary and systematic testing

A comparison of the results between developer test and evolutionary test showed that evolutionary testing found longer execution times for all the given modules. This is especially astonishing, because the Evolutionary Algorithm treats the software modules as black boxes.

If these promising results can be established during further optimizations, the developers of real-time software modules would gain an efficient tool for the determination of minimum and maximum execution times for their software modules. A lot of former testing by hand could be carried out automatically by evolutionary testing. The received results should be as good or better.

4 Concluding remarks

The correct functioning of real–time systems depends critically on their temporal correctness. Testing is the most important analytical method for the quality assurance of real-time systems.

In various experiments, evolutionary testing has been successfully applied to search for the longest and shortest execution times of real–time programs in order to check whether they violate specified timing constraints.

Evolutionary Algorithms show considerable promise in testing and validating the temporal correctness of real–time systems, and further research work in this area should prove fruitful. However, more work is still needed to find the most appropriate and robust parameters for evolutionary testing, so that the extreme execution times can be detected efficiently and with a high degree of certainty in practical operation.

However, the use of evolutionary algorithms alone is not sufficient for a thorough and comprehensive test of real–time systems. A combination with existing test procedures is necessary to develop an effective test strategy for embedded systems. A combination of systematic testing and evolutionary testing is promising [23].

If the evolutionary search does not start with a randomly generated population, but with a set of test data systematically determined by the tester, the disadvantage of evolutionary algorithms, namely that they might not find certain test relevant value combinations, can be compensated for. Moreover, evolutionary testing benefits from the tester's knowledge of the program function or structure.

In future, it is also intended to examine the combination with static analyses more closely [13]. By combining both approaches, the area in which one finds the extreme execution time of the system can be closely defined, e.g. static analyses give an upper estimate for the maximum execution time and testing gives a lower estimate for the maximum execution times.

References

[1] *Baker, J.E.*: Reducing Bias and Inefficiency in the Selection Algorithm. Proceedings of the Second International Conference on Genetic Algorithms and their Application, Cambridge, USA, 1987.

[2] *Beizer, B.*: Software Testing Techniques. New York: Van Nostrand Reinhold, 1990.

[3] *Davis, C.G.*: Testing Large, Real–Time Software Systems. Software Testing, Infotech State of the Art Report, Vol. 2, 1979, pp. 85–105, 1979.

[4] *Grochtmann, M., and Grimm, K.*: Classification Trees for Partition Testing. Software Testing, Verification & Reliability, Vol. 3, No. 2, pp. 63-82, 1993.

[5] *Grochtmann, M., and Wegener, J.*: Test Case Design Using Classification Trees and the Classification-Tree Editor CTE. Proceedings of Quality Week ’95, San Francisco, USA, 1995.

[6] *Grochtmann, M., and Wegener, J.*: Evolutionary Testing of Temporal Correctness. Proceedings of Quality Week Europe ’98, Brussels, Belgium, 1998.

[7] *Heath, W.S.*: Real-Time Software Techniques. Van Nostrand Reinhold, New York, USA, 1991.

[8] *Hennell, M.A., Hedley, D., and Riddell, I.J.*: Automated Testing Techniques for Real-Time Embedded Software. Proceedings of the European Software Engineering Conference ESEC ’87. Strasbourg, France, 1987.

[9] *Hetzel, B.*: The Complete Guide to Software Testing. Wellesley, MA: QED Information Sciences, 1988.

[10] *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, pp. 98-107, 1998.

[11] *Mathworks, The*: Matlab - UserGuide. Natick, Mass.: The Mathworks, Inc., 1994-1999. http://www.mathworks.com/

[12] *Mitchell, M.*: An Introduction to Genetic Algorithms. Cambridge, Massachusetts: MIT Press, 1996.

[13] *Mueller, F. and Wegener, J.*: A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints. Proceedings of the IEEE Real-Time Technology and Applications Symposium RTAS ’98, pp. 144-154, 1998.

[14] *Mühlenbein, H. and Schlierkamp-Voosen, D.*: Predictive Models for the Breeder Genetic Algorithm: I. Continuous Parameter Optimization. Evolutionary Computation. Vol. 1, No. 1, pp. 25–49, 1993.

[15] *Pohlheim, H.*: Genetic and Evolutionary Algorithm Toolbox for use with Matlab. http://www.geatbx.com/, 1994-1999.

[16] *Pohlheim, H.*: Entwicklung und systemtechnische Anwendung Evolutionärer Algorithmen. Aachen, Germany: Shaker Verlag, 1998. (Development and Engineering Application of Evolutionary Algorithms. Ph.D. thesis, in german)

[17] *Rational Software Corporation*: Quantify. Rational Software Corporation, 1997. http://www.rational.com/products/quantify/

[18] *Schultz, A.C., Grefenstette, J.J., and De Jong, K.A.*: Test and Evaluation by Genetic Algorithms. IEEE Expert. Vol. 8, No. 5, pp. 9-14, 1993.

[19] *Stankovic, J.A.*: Misconceptions about Real–Time Computing - A Serious Problem for Next–Generation Systems. IEEE Computer, Vol. 21, No. 10, pp. 10–19, 1988.

[20] *Sthamer, H.-H.*: The Automatic Generation of Software Test Data Using Genetic Algorithms. PhD Thesis, Department of Electronics and Information Technology, University of Glamorgan, Wales, UK, 1996.

[21] *Sullivan, M. O', Vössner, S. and Wegener, J.*: Testing Temporal Correctness of Real-Time Systems - A New Approach Using Genetic Algorithms and Cluster Analysis. Proceedings of EuroSTAR'98, pp. 397-418, 1998.

[22] *Wegener, J. and Pitschinetz, R.*: TESSY - Yet Another Computer–Aided Software Testing Tool? Proceedings of the Second European International Conference on Software Testing, Analysis & Review EuroSTAR ’94, 1994.

[23] *Wegener, J., and Grochtmann, M.*: Verifying Timing Constraints of Real-Time Systems by Means of Evolutionary Testing. Real-Time Systems, 15, pp. 275-298, 1998.

[24] *Whitley, D.*: The GENITOR Algorithm and Selection Pressure: Why Rank–Based Allocation of Reproductive Trials is Best. Proceedings of the Third International Conference on Genetic Algorithms, San Mateo, California, USA, pp. 116–121, 1989.