Quality Week '97

Testing Temporal Correctness of Real-Time Systems by Means of Genetic Algorithms

Joachim Wegener, Matthias Grochtmann

and

Bryan Jones

Daimler-Benz AG
Research and Technology
Alt-Moabit 96 a
D-10559 Berlin
Germany

Tel.: ++49 (0) 30 39982-{232, 229}
Fax: ++49 (0) 30 39982 107
E-mail: {wegener, grochtm}@DBresearch-berlin.de

University of Glamorgan
Department of Computer Studies
Pontypridd
CF37 1DL
UK

Tel.: ++44 (0) 1443 48 2730
Fax: ++44 (0) 1443 48 2715
Email: bfjones@glam.ac.uk

Abstract

The development of real-time systems is an essential industrial activity. Dynamic testing is the most important analytical method to assure the quality of real-time systems. It is the only method that examines the run-time behavior, based on an execution in the application environment.

An investigation of existing software test methods shows that they mostly concentrate on testing for functional correctness. They are not specialized in the examination of temporal correctness that is essential to real-time systems. Therefore, existing test procedures must be supplemented by new methods that concentrate on determining whether the system violates its specified timing constraints. In general, a violation means that outputs are produced too early or their computation takes too long. The task of the tester is to find the inputs with the longest or shortest execution times to check whether they produce a temporal error. If the search for such inputs is interpreted as a problem of optimization, genetic algorithms can be used to find the inputs with the longest or shortest execution times automatically. The fitness function is the measured execution time.

Experiments using genetic algorithms on a number of programs with up to 1511 LOC and 843 input parameters have successfully identified new longer and shorter execution times than those that had been found using random testing and systematic testing. Genetic algorithms therefore seem to be well-suited for checking the temporal correctness of real-time software. A combination of genetic optimization with systematic testing offers further opportunities to improve the test quality and could lead to an effective test strategy for real-time systems.

0 Introduction

Many industrial products use embedded computer systems. Usually, embedded computer systems have to fulfil real-time requirements, and correct system functionality depends on their logical correctness as well as on their temporal correctness.

In practice dynamic testing is the most important analytical method for assuring the quality of embedded computer 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. Often more than 50 % of the overall development budget is spent on testing [Davis, 1979].

For testing real-time systems the examination of the functional system behavior alone is not sufficient. Additionally, the temporal behavior of the systems needs to be thoroughly examined. An investigation of existing software test methods shows that they mostly concentrate on testing for functional correctness. They are not suited for an examination of temporal correctness which is also essential to real-time systems. This work tries to fill this gap by giving support to testing temporal behavior. It investigates the effectiveness of genetic algorithms to validate the temporal correctness of embedded systems by establishing the maximum and minimum execution times. Promising results have been achieved. However, the sole use of genetic algorithms is not sufficient for a thorough and comprehensive examination 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 genetic optimization is promising.

The first section contains an overview of the current state in the field of testing real-time systems. The second section gives a brief introduction to genetic algorithms and describes how they are applied to solve different testing problems. This is followed by a depiction of the way genetic algorithms are used for testing the temporal behavior of real-time systems. Several experiments were performed. Their results will be described in detail. Section 4 discusses the combination of systematic testing and genetic optimization and derives from it a test strategy for real-time systems. After some concluding remarks the paper closes with a short outlook on current and future work.

1 Testing Real-Time Systems

Testing is one of the most complex and time-consuming activities within the development of real-time systems [Heath, 1991]. It typically consumes 50 % of the overall development effort and budget since embedded systems are much more difficult to test than conventional software systems. 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.

Real-time systems must be tested for compliance with their functional specification and their timing constraints. An investigation of existing software test methods shows that a number of proven functional and structural test methods is available for examining logical correctness, e.g. the classification-tree method [Grochtmann and Grimm, 1993]. When using structure-oriented test methods the tester must take into account that an instrumentation of the test object causes probe effects, i.e. deviations from the real system behavior are possible. There are no specialized procedures for testing temporal behavior. This is why in practice testers often go back to conventional test procedures. They try to compensate for the existing methodological shortcomings by using more or less heuristic tests in worst-case scenarios. However, even a small experiment makes clear that testing temporal behavior is a very complex task which requires special methods and tools.

1.1 Experiment

For the experiment, a simple computer graphics function written in C was thoroughly tested. It contained a total of 37 statements and had 16 program branches; its control flow graph is shown in figure 1. A systematic test with the classification-tree method led to 49 test cases which covered all the branches and resulted in 43 different execution times. The timings varied between 359 processor cycles (equivalent to 5.27 ms) and 1839 processor cycles (equivalent to 26.27 ms).

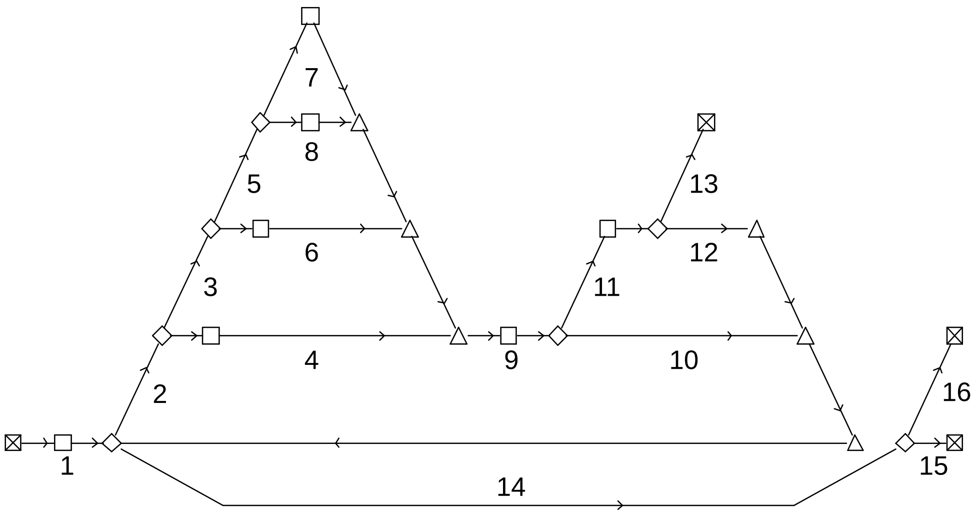


Figure 1: Control Graph

This was followed by applying a total of 4603 randomly generated tests, which resulted in 298 different execution times even though full branch coverage was not achieved; one branch remained untested. The timings varied again between 359 cycles and 1839 cycles.

These initial results demonstrate the importance of investigating a specialized approach to testing temporal correctness: already very small systems show a wide range of different execution times. Only a fraction of the possible execution times is covered by the systematic test. Random testing, however, does not detect certain value combinations that might be significant for the temporal behavior. Consequently, for testing temporal behavior the application of a new approach was examined, namely genetic algorithms.

2 Genetic Algorithms

Genetic algorithms are a well-established method of optimization in many areas. An overview of different successful applications is, for example, provided by Davis [1996]. Genetic algorithms represent a class of adaptive search techniques and procedures based on the processes of natural genetics and Darwin's theory of evolution. Genetic algorithms model natural processes, such as selection, reproduction, mutation, migration, locality, and neighborhood [Pohlheim, 1996]. The fundamental concept of genetic algorithms is to evolve successive generations of increasingly better combinations of those parameters which significantly effect the overall performance of a design. The genetic algorithm achieves the optimum solution by the random exchange of information between increasingly fit samples (combination/crossover) and the introduction of a probability of independent random change (mutation). Traditionally, parameters involved in genetic optimization have been represented as strings of binary bits where crossover is achieved by choosing a point along two bit strings at random and swapping the tails, and mutation by picking a bit at random and flipping its value. The adaptation of the genetic algorithm is achieved by the selection and survival procedures since these are based on fitness. The fitness-value is a numerical value that expresses the performance of an individual with regard to the current optimum so that different designs may be compared. The notion of fitness is fundamental to the application of genetic 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.

Figure 2 gives an overview of a typical procedure for genetic algorithms. A population of guesses to the solution of a problem is initialized, usually at random. Each individual in the population is evaluated by calculating its fitness. This will result in a spread of solutions ranging in fitness from very poor to good - the chances of hitting on the optimum solution initially are, of course, infinitesimally small for most problems. The remainder of the algorithm is iterated until the optimum is achieved. Pairs of individuals are selected from the population *using a pre-defined strategy, and are combined in some way to produce a new guess in an analogous way to biological reproduction. Combination algorithms are many and varied. Additionally, mutation is applied. The new individuals are evaluated for their fitness, and survivors into the next generation are chosen from the parents and offspring, often according to fitness though it is important to maintain a diversity in the population to prevent premature convergence to a sub-optimal solution.*

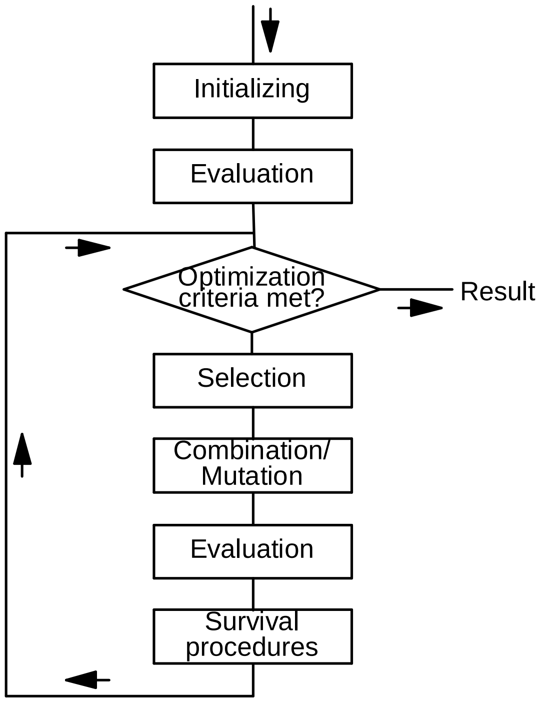


Figure 2: Block Diagram of Genetic Algorithms

2.1 Genetic Algorithms Applied to Testing

Genetic algorithms have been applied successfully to various testing problems. Several papers deal with structural testing; others concentrate on test case generation based on formal specifications, the testing of APIs, and testing for robustness.

Jones et al. [1996] have used genetic algorithms to generate test data automatically to execute every branch in a variety of programs written in Ada83. In most cases full branch coverage was obtained. The branch predicate formed the basis of the fitness function so that boundary value data were generated.

Roper [1996] has obtained encouraging results by applying genetic algorithms to achieve branch coverage for programs written in C or C++. The test object is instrumented with probes to provide feedback on the coverage achieved. For each individual, the program path executed determines its level of fitness.

Xanthakis et al. [1992] describe a method in which a constraint propagation graph is formed; the nodes of the graph may represent either predicates or variables connected by elementary path functions. The fitness function depends on the predicates which are satisfied by modifying the adjacent variable nodes by a small amount. This work was continued by Watkins [1995] who used a fitness function based on the reciprocal of the number of times a path was exercised.

Jones et al. [1995] have derived test sets from Z specifications by a method that used a variety of algorithms, including genetic algorithms and simulated annealing. A language was developed to enter the Z schemas into the machine and a series of test cases was formed for both valid and invalid inputs.

Boden and Martino [1996] have developed a testing facility in which genetic algorithms are used to generate API tests. The fitness function is a weighted sum of various factors of a test response, e.g. depending on the generation of exceptions, well-defined errors or return codes by the API. Furthermore, sequences of API calls are determined as useful, based on expected or recommended usages for the API.

Schultz et al. [1993] have achieved promising results using genetic algorithms for testing the robustness of autonomous vehicle controllers. The aim of the test was to find test scenarios in which minimal fault activity causes a mission failure or vehicle loss and in which maximal fault activity still permits a high degree of mission success. These scenarios provided some insight into parts of the controller and allowed the designer to improve the controller's robustness. The fitness function was based on the current fault activity and the quality of mission fulfilment.

In this work, we investigate the feasibility of genetic algorithms for testing the temporal correctness of real-time systems.

3 The Application of Genetic Algorithms to Testing Temporal System Behavior

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. 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. This search for the shortest and longest execution times can be regarded as an optimization problem to which genetic algorithms seem an appropriate solution.

Genetic algorithms enable a totally automated search for the longest and shortest execution times. They are particularly suited to problems involving large numbers of variables and complex input domains. Even for non-linear and poorly understood search spaces genetic algorithms have been used successfully. Since genetic algorithms search from a population of points rather than from a single point, the probability of getting stuck at local optima is significantly reduced compared with more traditional optimization techniques, like hill climbing. The use of mutation and subpopulations can further reduce the chance of getting stuck. When genetic algorithms are used to solve optimization problems, good results are obtained surprisingly quickly [Sthamer, 1996].

Figure 3 illustrates the use of genetic optimization for determining the shortest and longest execution times. The initial population is generated at random. Each individual of the population represents a test datum with which the test object is executed. For every test datum the execution time is measured. The execution time determines the fitness of the respective individual or test datum. If one searches for the longest execution time, test data with long execution times obtain high fitness values. If one searches for the shortest execution times, individuals with short execution times obtain high fitness values. Afterwards, members of the population are selected with regard to their fitnesses and subjected to combination and mutation to generate a new population. First, it is checked whether the generated test data are in the input domain of the test object. Then the individuals of the new generation are also evaluated and united with the previous generation to form a new population according to the survival procedures laid down. Afterwards, this process repeats itself, starting with selection, until a given stopping condition is reached or a temporal error is detected. Thus an execution time is found which is outside the specified timing constraints. 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.

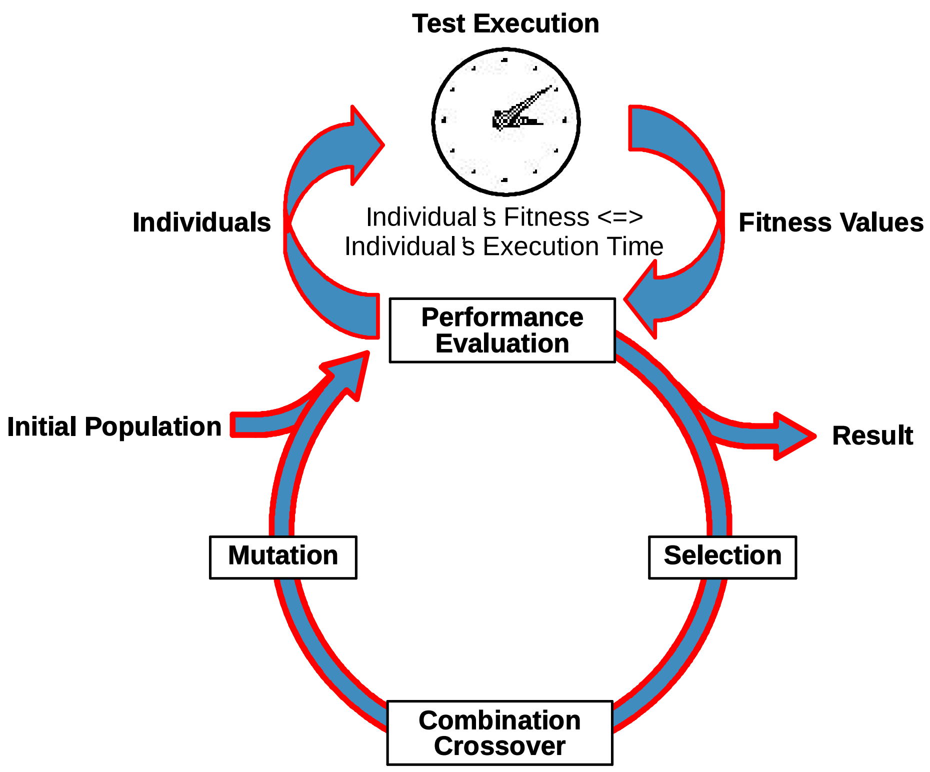


Figure 3: Closed-Loop Optimization of Execution Times

3.1 Experiments

We used genetic algorithms in several experiments to determine the shortest and longest execution times of different systems. Of course the results are dependent on the hardware/software platform and generally are not directly transferable from one to another since the processor speed and the compiler used directly affect the temporal behavior. All the experiments to be described were carried out on a SPARCstation 10 running under Solaris 2.5. The duration of executions was measured in processor cycles to rule out overheads by the operating system such as service interrupts. Thus the execution times reported were the same for repeated runs with identical parameters.

The fitness was set equal to either the execution time for the longest path or its reciprocal for the shortest path measured in processor cycles. For each experiment the genetic algorithms were applied twice, first, to find the longest execution time, and then the shortest. The results of the experiments are summarized and compared to random testing in table 1. Two libraries of genetic algorithms were used - one was a Matlab-based toolbox developed at the Daimler-Benz Laboratories by Hartmut Pohlheim and the second was a harness developed in Ada83 by Harmen Sthamer at the University of Glamorgan.

When genetic algorithms were applied to testing the simple C function already mentioned in section 1.1, the longest execution time of 1839 cycles was found in less than 20 generations and a new shortest time of 355 cycles (5.07 ms*) was discovered in a smaller number of tests than executed for random testing (see Computer Graphics II in table 1). The validity of the longest and the new fastest path found by the genetic algorithms was verified by analyzing the control flow graphs printed in figure 4, though this would be difficult to achieve in general for large complex software. For the shortest execution time only the path 1-2-4-9-11-13 is executed; for the longest execution time all branches are executed except the empty ones numbered 10, 13, and 15.*

Applications	*Random Testing*			*Genetic Optimization*		
	No. of Test Runs	*Shortest Exec. Time*	*Longest Exec. Time*	*No. of Test Runs*	*Shortest Exec. Time*	*Longest Exec. Time*
Computer Graphics I (61 LOC)	*9200*	*99*	*384*	*1200*	*99*	***392***
Computer Graphics II (107 LOC)	*4600*	*359*	*1839*	*800*	***355***	*1839*
Auto Electronics I (432 LOC)	*2700*	*66*	*104*	*1350*	***45***	*104*
Auto Electronics II (1511 LOC)	*5000*	*366*	*10774*	*4850*	*366*	***12185***
Railroad Technology (389 LOC)	*10000*	*1050*	*11529*	*9500*	***399***	***14175***
Defense Electronics (879 LOC)	*56000*	*27258*	*110842*	*30000*	***27154***	***114160***

Table 1: Shortest and Longest Execution Times in Processor Cycles Measured for a Variety of Programs by Random Testing and Genetic Algorithms; the Overall Optimum is Shown in Bold

The first computer graphics example (Computer Graphics I) shows that even when the number of test runs for random testing was multiplied, compared with the number of test runs for genetic optimization, the longest execution time determined by the genetic algorithms could not be found by random testing. For the second computer graphics example and the first example from the field of automotive electronics shorter execution times were found by the genetic algorithms than by random testing.

The second automotive electronics system implements the entire functionality of an airbag controller and therefore is safety critical. It contains 1511 LOC. For this example about 80 parameters were varied by the genetic algorithms, among other things the interval between the occurrence of different events.

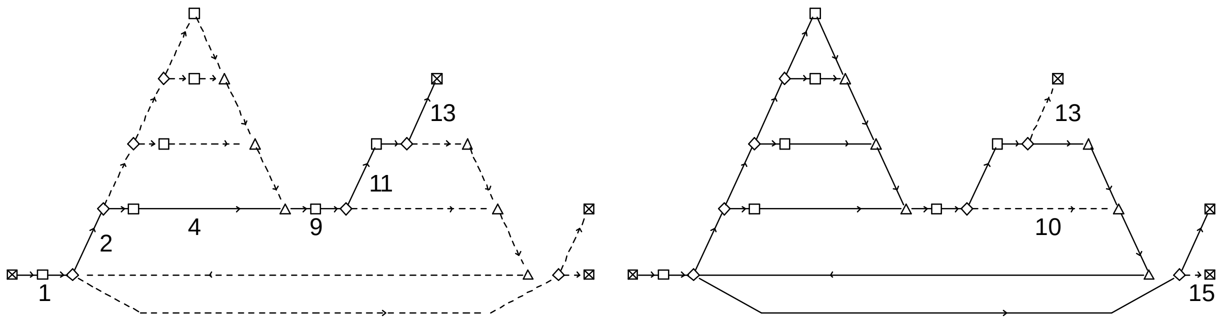


Figure 4: Control Flow Graphs for the Shortest (5.07 msec) and the Longest (26.27 msec) Execution Time of Computer Graphics II

The genetic algorithm used in this case was typical of all our experiments. The population contained 50 test sets. Pairs of tests were chosen at random and combined using a double crossover algorithm, i.e. the middle part of the bit strings is exchanged. The mutation probability was set to 0.00345. During survival, test sets were chosen from both the parent population with a probability of 90 % and from the offspring with 10 %; the next generation therefore contained fewer offspring than parents and effectively slowed the rate of change to maintain stability of the algorithm. There is no means of deciding when an optimum path has been found and the genetic algorithms were allowed to continue for 100 generations before they were stopped.

When searching for the longest execution time of the airbag controller software, a maximum of 12185 processor cycles was found in generation number 97. In the same way, the random test was terminated after 5000 tests, and at this time had found a maximum of 10774 cycles. The genetic algorithms had found an execution time 13% longer than was found for random testing. Figure 5 shows that random testing reaches its maximum execution time after only about 1500 test runs, whereas for genetic optimization a continuous improvement up to the 100th generation can be observed. The curve trace suggests that the genetic algorithms would find even longer execution times if the number of generations were increased. After only eleven generations - that corresponds to 550 test runs - the execution times found by genetic optimization are above those of random testing.

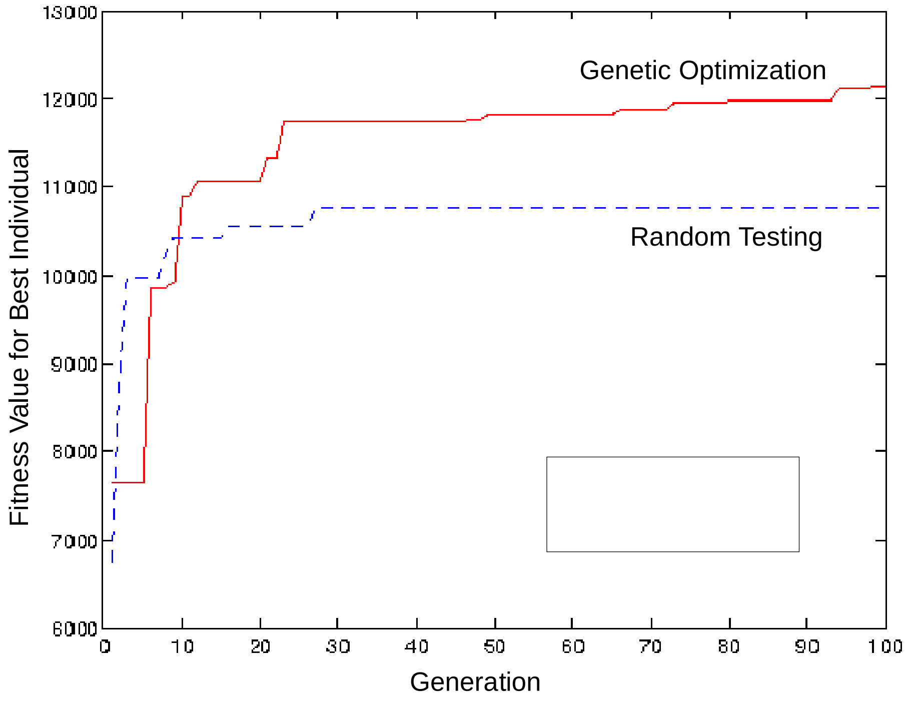


Figure 5: Comparison of Genetic Optimization and Random Testing Searching for the Longest Execution Time for the Airbag Controller

When searching for the shortest execution time, both genetic algorithms and random testing found the same time of 366 cycles. The genetic algorithms discovered this in the first generation at which point it is still effectively a random search since the crossover and mutations could have had no effect. This path was clearly one whose tests occupied a large input subdomain with a high probability of being found at random.

This is also emphasized by Figure 6 which indicates the frequency with which different execution times occurred during the search for the longest execution time. For random testing far more than 10 % of all test runs goes to particularly short run times. For the other execution times almost a Gaussian curve results from random testing. During genetic optimization, however, not even 1 % goes to the area of particularly short run times. Furthermore, a clearly increasing number can be seen for the long execution times. The genetic algorithms obviously succeed in avoiding the generation of test data with short run times and in concentrating on test data with long execution times.

The railroad control and instrumentation technology example is also safety critical. The population size in this experiment was increased to 100 because of the complexity of the test object. The shortest execution time found by the genetic algorithms is more than 60 % below the one detected by random testing. 14175 processor cycles were determined as the longest execution time by the genetic algorithms. This is 23 % above the maximum execution time of 11529 cycles that was found by random testing. Figure 7 shows the comparison of random

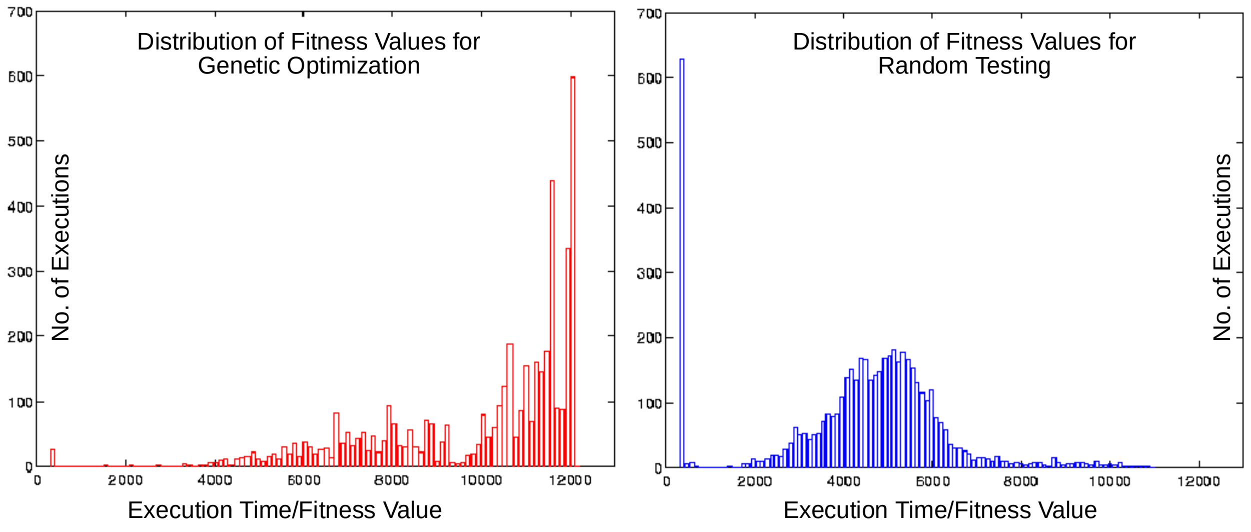


Figure 6: Distribution of Execution Times for Genetic Optimization and Random Testing when Searching for the Longest Execution Time for the Airbag Controller

testing and genetic optimization for the search of the longest execution time. It becomes clear after the fourth generation that genetic optimization is superior to random testing. Random testing stagnates after 3500 test runs; the generation of 6500 other test data sets does not produce any longer execution times. Genetic optimization, however, manages once again to improve the execution times continuously. In generation number 70 even a significant leap of more than 1500 cycles can be noted.

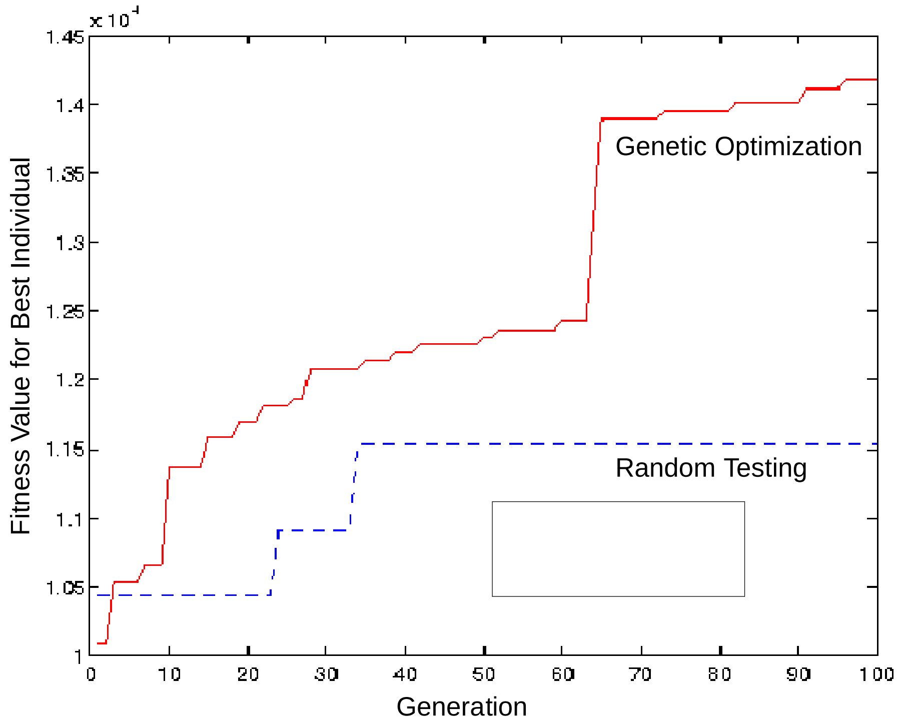


Figure 7: Comparison of Genetic Optimization and Random Testing Searching for the Longest Execution Time for the Railroad System

The Defense Electronics program with 879 LOC has 843 integer input parameters. The first two input parameters represent the position of a pixel in a window and lie within the range 1..1200 and 1..287 respectively. The remaining 841 parameters define an array of 29 by 29 pixels representing a graphical input located around the specified position; each integer describes the pixel color and lies in the range 0..4095. Genetic algorithms were used in this example to generate pictures surrounding a given position. The longest execution time increased steadily with each new generation and asymptoted towards the current maximum of 114160 cycles when the run was terminated after 300 generations. The population size in this experiment was also set to 100 because of the large range of the variables and the large number of input parameters. The fastest execution time was found to be 27154 after 100 generations. The genetic algorithms found more extreme values for the longest and shortest execution times than those found by random testing that were 110842 and 27258 respectively. Where genetic algorithms allowed to search further a longest execution time of 114393 processor cycles was found in generation 657 and a shortest execution time of 26814 was detected in generation 372.

3.2 Discussion

In all our experiments genetic algorithms obtained better results than random testing, regardless of whether the shortest or the longest execution times were searched for. The disadvantage of a random method which is that no step builds upon another is avoided by using genetic algorithms. Genetic algorithms take advantage of the old knowledge held in a parent population to generate new guesses with improved performance. Their iterations are based on the experience which has been gained from previous trials.

In most experiments the shortest execution times were found more quickly than the longest execution times. Previous experience with genetic algorithms and random testing suggests that this is due to the shorter execution times having a large input subdomain which has a higher probability of being found at random. The shorter execution times will usually be associated with short control flow paths. As the control flow path increases, there tends to be a filtering effect arising from a sequence of predicates which reduces the probability of control passing along any path through the broadening tree of paths. Consequently, the input subdomains associated with longer paths tend to be smaller and therefore have a smaller probability of being found at random. Genetic algorithms come into their own in such circumstances and are able to find appropriate test sets with less effort than random testing [Jones et al., 1996].

In several experiments local searches, like simulated annealing, were used in an attempt to improve on the results from genetic algorithms, but without success. This is probably because the fitness for large subsets of the input domain is constant and local searching detects no improvement. The optimal solution sought represents an isolated and small subdomain and is best found by sampling the input domain widely.

For both computer graphics examples systematic tests with the classification-tree method were also performed. The longest execution time of 392 processor cycles found by the genetic algorithms for the first example was detected by none of the systematic tests. Only 384 cycles as maximum and 99 cycles as minimum run time were found by the systematic tests. For the second example systematic testing and genetic optimization detected the same run times, namely 355 and 1839 processor cycles. In some simpler experiments systematic testing turned out to be superior to genetic optimization. For a sorting program, for example, the list sorted in reverse order caused the longest run time. This was steadily approximated by the genetic algorithms but not yet reached after 100 generations.

Compared with static analysis, testing by means of genetic optimization overcomes relevant problems of static analysis, like the consideration of pipelining and caching [Healy et al., 1995] as well as delays caused by direct memory access (DMA) or DRAM refresh cycles [Müller, 1996]. A further advantage is that the test object is executed in the real application environment. Moreover, genetic optimization can in principle also be applied for determining the shortest and longest execution times of parallel and distributed systems where static analyses currently come up against limiting factors.

Since genetic algorithms try to achieve the optimum solution by the random exchange of information between increasingly fit samples (combination) and the introduction of independent random change (mutation), they *share a problem with random and statistical testing: it is not predictable if and when certain input situations will be found, which might be especially important for the run-time behavior of the system under test. Therefore, we cannot prove that the timings found are the longest and shortest possible values. On the other hand, existing approaches to systematic testing are not sufficient to examine the temporal behavior of systems thoroughly. Consequently, an effective test strategy for real-time systems should contain systematic testing as well as genetic optimization.*

4 Test Strategy

As a strategy for testing real-time systems we recommend the combination of systematic testing with genetic optimization. By means of the systematic test errors in the logical program behavior of the test object shall be detected. Furthermore, special value combinations shall be defined which are relevant to the testing of temporal behavior but which might be difficult to find with the help of genetic algorithms. On the basis of the systematic test genetic algorithms are then used to detect input constellations with particularly long and short run times which the tester did not find by means of the systematic test.

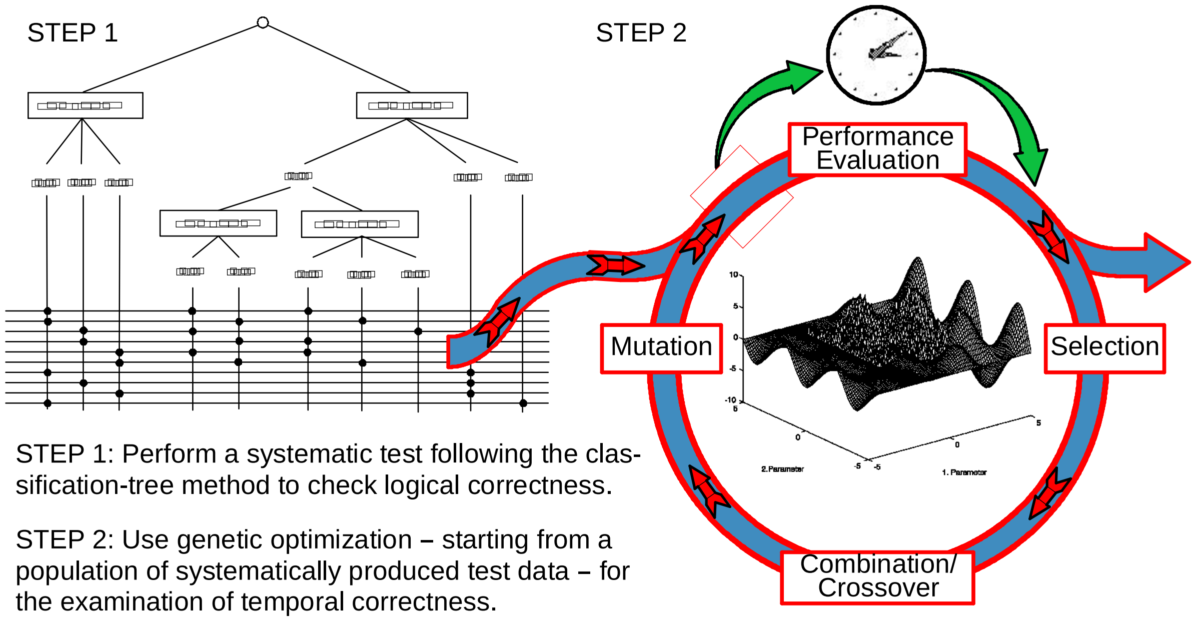
The classification-tree method should be applied for the systematic test because it is a functional test method which has already proved very worthwhile in practice (cf. [Grochtmann and Wegener, 1995]). The function-oriented test is indispensable to the thorough examination of systems [Grimm, 1996] for only by means of test cases derived from the system specification can it be found out if specified requirements or functions were omitted (e.g. simply forgotten) during the software development process.

The test strategy comprises two steps. At first, the tester uses the classification-tree method for the systematic design of black-box test cases. The tester also adds aspects assessed as relevant to the temporal system behavior, for example the simultaneous occurrence of several events or time-consuming system states. However, test cases determined with the classification-tree method focus mainly on the examination of logical correctness. Afterwards, the second step of our test strategy concentrates on the examination of temporal correctness. The test data specified for the systematic test is used as initial population for the optimization of execution times by means of genetic algorithms - as described in section 3. Thus the genetic search for the shortest and longest execution times benefits from the tester's experience and his domain knowledge [Wegener et al., 1996]. Figure 8 illustrates the suggested test strategy for real-time systems.

In principle structure-oriented test cases are also suited as initial population for the genetic search because there is a close correlation between temporal behavior and program structure. The number of processor cycles measured will generally be directly related to the number of statements in the control flow path, though there will be exceptions because some statements require more cycles than others [Wegener et al., 1997]. In this case genetic algorithms will benefit from the tester's knowledge of the internal program structure. Another idea for further improvement is to link genetic optimization directly with structural testing. The fitness-function could be expanded in such a way that individuals which execute a new program branch or path would get a high fitness-value to ensure their survival in the next generation. Thus the diversity of the population would not only be maintained with respect to the temporal behavior of individuals but also in consideration of the test object's internal structure. From this follows that the new program structures would be executed several times in the next generations. If no longer execution times resulted from this, the corresponding test sets would become extinct again during subsequent generations.

5 Conclusion and Future Work

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 such systems. An investigation of existing testing approaches showed a lack of support for testing the temporal behavior. Therefore, existing test procedures must be supplemented by new methods and tools. In various experiments genetic algorithms have been successfully applied to search the longest and shortest execution times of real-time programs in order to check



STEP 1: Perform a systematic test following the classification-tree method to check logical correctness.

STEP 2: Use genetic optimization – starting from a population of systematically produced test data – for the examination of temporal correctness.

Figure 8: Test Strategy Suggested for Real-Time Systems

whether they violate the specified timing constraints. Compared with random testing, genetic algorithms always obtained better results.

Further improvements are possible through the combination with systematic test methods. If the genetic search does not start with a randomly generated population but with a set of test data systematically determined by the tester, the disadvantage of genetic algorithms that they might not find certain test relevant value combinations can be compensated for. Moreover, depending on the test method applied, genetic optimization benefits from the tester's knowledge of the program function or the program structure.

Genetic 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. More work is needed to find the most appropriate parameters for genetic algorithms and to define suitable criteria for the decision when to stop the search. Further studies are focusing on the question how stagnations can be reacted to with appropriate changes of the search strategy.

In the future it is also intended to examine the combination with static analyses more closely. If static analyses can be applied, they usually give an upper estimate for the maximum run time. Genetic optimization, however, produces execution times that were actually measured. It cannot be guaranteed though that these are the longest possible execution times. Hence, genetic optimization gives a lower estimate for the maximum run time. By combining both approaches the area in which one finds the maximum run time of the system can thus be closely defined. This also makes clear that the possible applications of genetic optimization are not confined to testing real-time systems. Using genetic optimization is also appropriate for the computation of worst-case execution times of hard real-time tasks during the design of scheduling strategies when static analyses are out of the question for this.

References

Boden, E.B., and Martino, G.F. (1996). Testing Software Using Order-Based Genetic Algorithms. In Koza, J.R. et al. (eds.). Proceedings of the First Annual Conference on Genetic Programming, 28 - 31 July 1996, Stanford University. The MIT Press, Cambridge, USA, pp. 461 - 466.

Davis, C.G. (1979). Testing Large, Real-Time Software Systems. Software Testing, Infotech State of the Art Report, Vol. 2, 1979, pp. 85 - 105.

Davis, L. (1996). Handbook of Genetic Algorithms. International Thomson Computer Press, Boston, USA.

Grimm, K. (1996). Systematic Testing of Software-Based Systems. Proceedings of the 2nd Annual ENCRESS Conference, June 1996, Paris, France.

Grochtmann, M., and Grimm, K. (1993). Classification Trees for Partition Testing. Software Testing, Verification & Reliability, Vol. 3, No. 2, pp. 63 - 82, Wiley.

Grochtmann, M., and Wegener, J. (1995). Test Case Design Using Classification Trees and the Classification-Tree Editor CTE. Proceedings of Quality Week '95, 30 May - 2 June 1995, San Francisco, USA.

Healy, C.A., Whalley, D.B., and Harmon, M.G. (1995). Integrating the Timing Analysis of Pipelining and Caching. IEEE Real-Time Systems Symposium, pp. 288 - 297, 4 - 7 December 1995, Pisa, Italy.

Heath, W.S. (1991). Real-Time Software Techniques. Van Nostrand Reinhold, New York, USA.

Jones, B.F., Sthamer, H.-H., and Eyres, D.E. (1996). Automatic Structural Testing Using Genetic Algorithms. Software Engineering Journal, Vol. 11, No. 5, pp. 299 - 306, IEE & BCS, Stevenage, UK.

Müller, F. (1996). Statische Analyse der maximalen Programmausführungszeiten (Static Analysis of the Maximum Program Execution Times). Lecture given at Daimler-Benz AG, Research and Technology, September 1996, Berlin, Germany.

Pohlheim, H. (1996). GEATbx: Genetic and Evolutionary Algorithm Toolbox for Use with Matlab – Documentation. Technical Report, Technical University Ilmenau.

Roper, M. (1996). CAST with GAs – Automatic Test Data Generation via Evolutionary Computation. Computer Aided Software Testing (CAST) Tools, IEE Colloquium C6, Digest No. 96/096.

Schultz, A.C., Grefenstette, J.J., and De Jong, K.A. (1993). Test and Evaluation by Genetic Algorithms. IEEE Expert, Vol. 8, No. 5, pp. 9 - 14, IEEE Computer Society.

Sthamer, H.-H. (1996). The Automatic Generation of Software Test Data Using Genetic Algorithms. PhD Thesis, Department of Electronics and Information Technology, University of Glamorgan, Wales, UK.

Watkins, A.E.L. (1995). A Tool for the Automatic Generation of Test Data Using Genetic Algorithms. Proceedings of Software Quality Conference '95, July 1995, Dundee, Scotland.

Wegener, J., Grimm, K., Grochtmann, M., Sthamer, H.-H., and Jones, B.F. (1996). Systematic Testing of Real-Time Systems. Proceedings of EuroSTAR '96, 2 - 6 December 1996, Amsterdam, Netherlands.

Wegener, J., Sthamer, H.-H., Jones, B.F., and Eyres, D.E. (1997). Testing Real-Time Systems Using Genetic Algorithms. Proceedings of Software Quality Management '97, 24 - 26 March 1997, Bath, UK.

Xanthakis, S., Ellis, C., Skourlas, C., LeGall, A., and Katsikas, S. (1992). Application of Genetic Algorithms to Software Testing. 5th International Conference on Software Engineering, December 1992, Toulouse, France.