EuroSTAR'98

Testing Temporal Correctness of Real-Time Systems
— A New Approach Using Genetic Algorithms and Cluster Analysis —

Michael O'Sullivan, Siegfried Vössner

and

Joachim Wegener

Stanford University
Department of Engineering Economic Systems
and Operations Research
Stanford, CA 94305-4023
U.S.A.

Telephone: +1 650 723 5576
Fax: +1 650 723 4107
{mosu, voessner}@leland.stanford.edu

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

Telephone: +49 (0) 30 39982 232
Fax: +49 (0) 30 39982 107
Joachim.Wegener@dbag.bln.daimlerbenz.com

Abstract

Testing is one of the most complex and time-consuming activities within the development of embedded systems. Usually, embedded systems have to fulfil real-time requirements, and correct system functionality depends on both their logical correctness and their temporal correctness. An investigation of existing testing approaches shows a lack of support for testing temporal system behavior. For this reason, existing test procedures must be supplemented by new methods that concentrate on determining whether or not the system violates its specified timing constraints. In most cases, a violation occurs because outputs are produced too early, or their computation takes too long. The tester must therefore 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, genetic algorithms can be used to automatically find the inputs with the longest or shortest execution times. The automatic search for accurate test data by means of genetic algorithms is called evolutionary testing.

Evolutionary testing is an iterative testing procedure based on the use of genetic algorithms, which approximates the best-case and worst-case execution times of the system under test within several generations. The test is regarded as a discontinuous, nonlinear, optimization problem, with the input domain of the test object as search space, sets of test data as decision variables, and execution times as objective values.

An important point of evolutionary testing is to decide when a test is to be terminated. We have therefore extended our approach with a technique based on cluster analysis. This technique examines the distribution of the individuals of a generation in the search space. Individuals usually form clusters in the search space after several iterations. These clusters provide information about the convergence state of the optimization run. This aids in forming a reliable termination criterion. Cluster analysis also provides additional information about local optima in order to detect several performance leaks within one test run. This is a good basis for optimizing the program code. Furthermore, cluster analysis gives information about the structure of the entire search space — something that might be unknown if the source code of the test object is too complicated to be analyzed or not available (e.g. in case of black-box testing). The use of cluster analysis will be illustrated by several experiments.

0 Introduction

The development of real-time systems is an essential industrial activity. The correct system functionality of real-time systems 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 real-time systems. Dynamic testing is the only method that examines the run-time behavior of systems, based on an execution in the application environment. 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 for embedded software is spent on testing.

Most existing test methods focus on testing for logical correctness as they are not specialized in the examination of temporal correctness, which is also essential to real-time systems. For this reason, existing test procedures must be supplemented by new methods, which concentrate on determining whether the system violates its specified timing constraints. This work tries to fill the gap by giving support to testing temporal behavior. A temporal error is normally caused because outputs are produced too early, or the computation of the outputs takes too long.

The tester must therefore 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, genetic algorithms can be used to automatically find the inputs with the longest or shortest execution times. This automatic search for accurate test data by means of genetic algorithms is called evolutionary testing.

Evolutionary testing examines the temporal behavior of a system as a discontinuous, nonlinear, optimization problem, with the input domain of the system under test as search space, sets of test data as decision variables, and execution times as objective values. Evolutionary testing is an iterative testing procedure based on genetic algorithms. Unlike classical optimization techniques, genetic algorithms use the principles of an apparently very powerful optimization tool — biological evolution. They imitate fundamental principles like selection, recombination and mutation. The concept is to evolve successive generations of increasingly better combinations of those parameters which significantly effect the overall performance of a design. Starting with a selection of good samples, the genetic algorithm achieves the optimum solution by the random exchange of information between these increasingly fit samples (recombination) and the introduction of a probability of independent random change (mutation). The

adaptation of the genetic algorithm is achieved by selection and reinsertion procedures since these are based on fitness values which permit an assessment of the performance of the samples.

When using evolutionary testing for the examination of real-time systems' temporal behavior, the main objective is to find test data which produce execution times violating the timing constraints specified. If a temporal error is found, the test was successful. In this case the test can be terminated. If, however, no errors in the temporal behavior have been detected after a succession of iterations, 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, recent work extends our approach with a technique based on cluster analysis, which enables us to assess the convergence of the generations.

With this extension we address three important issues in testing real-time systems. First, the clusters give more information about the structure of the search space — something entirely unknown if the source code of the test object is too complicated to be analyzed or not available. Second, the local optima are found from the clusters — using only short test runs. These optima are useful when optimizing the program code and are found without a large amount of additional work. Third, the clusters add information about the convergence state of the optimization run. This aids in forming a reliable termination criterion, which seems to be of even higher importance.

The first section of our paper gives an overview of the current state of testing real-time systems in practice. The second section deals with evolutionary testing and describes how it is applied to examine the temporal behavior of real-time systems. The third section introduces cluster analysis. The fourth section explains termination criteria usually used to end a genetic algorithm run. Subsequently, section 5 describes the combination of evolutionary testing and cluster analysis. Several experiments comparing cluster analysis with other termination criteria were performed. Their results will be described and discussed in detail. Furthermore, a strategy to detect local optima on the basis of cluster information will be presented. After some concluding remarks the paper closes with a short outlook on 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 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 [Grochtmann and Grimm, 1993], which has already been used with success for the functional test of different real-time systems from various application fields [Grochtmann and Wegener, 1995]. 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 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 [Hennell et al., 1987]. Schütz [1993] discusses suggestions how probe effects can be avoided.

For examining temporal correctness there are no specialized test methods available which are suited for industrial use [Wegener and Grochtmann, 1998]. For this reason, we have developed and examined a new approach for testing temporal behavior which is based on the use of genetic algorithms, namely evolutionary testing.

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. 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.

2.1 Brief Introduction to Genetic Algorithms

Genetic algorithms [Goldberg, 1989] 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 in parallel on a number of potential solutions, the population of individuals. In every individual, permissible solution values for the variables of the optimization problem are coded.

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. Starting with a selection of good individuals, the evolutionary algorithm achieves the optimum solution by the random exchange of information between these increasingly fit samples (recombination) and the introduction of a probability of independent random change (mutation). The adaptation of the genetic

algorithm is achieved by the selection and reinsertion procedures used because these are based on fitness. The 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 the offspring population to form the next generation. The fitness-value is a numerical value that expresses the performance of an individual with regard to the current optimum so that different 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 1 gives an overview of a typical procedure for genetic algorithms. First, 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. Usually, this will result in a spread of solutions ranging in fitness from very poor to good. The remainder of the algorithm is iterated until the optimum is achieved, or another stopping condition is fulfilled. Pairs of individuals are selected from the population according to the pre-defined selection strategy, and 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 diversity in the population to prevent premature convergence to a sub-optimal solution.

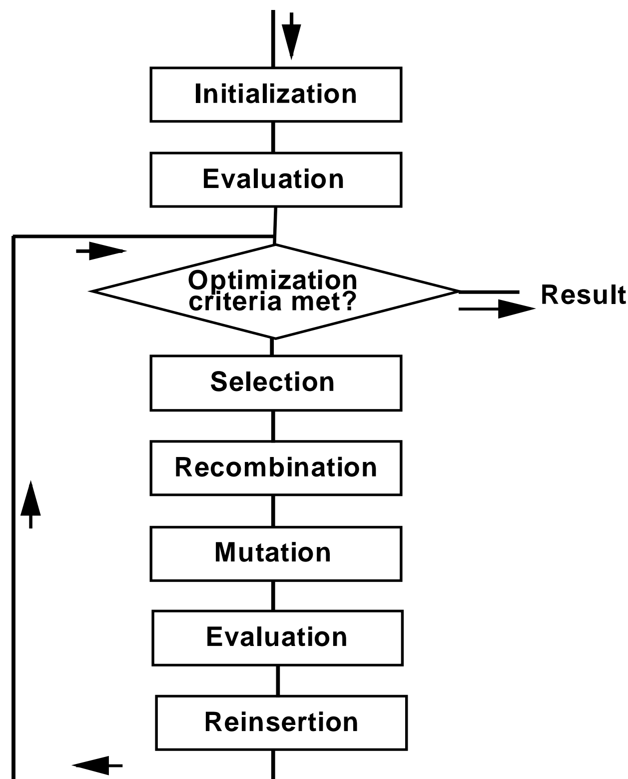


Figure 1. Block Diagram of Genetic Optimization

Genetic algorithms 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 [Sthamer, 1996].

Genetic algorithms have been successfully applied to various testing problems. Several papers deal with structural testing (e.g. [Jones et al., 1998] and [Roper, 1997]), others concentrate on test case generation based on formal specifications [Jones et al., 1995], the testing of APIs [Boden and Martino, 1996], and testing for robustness [Schultz et al., 1993]. Wegener et al. [1997] offer a brief survey on applications of evolutionary testing.

2.2 Application of Genetic Algorithms to Testing Temporal Behavior

In this work, genetic algorithms are used for testing the temporal correctness of real-time systems. Figure 2 illustrates how evolutionary testing proceeds to determine the best-case and worst-case 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 worst-case execution time, test data with long execution times obtain high fitness values. If one searches for the best-case execution time, 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 that the generated test data are in the input domain of the test object. Then the individuals of the new generation are also evaluated and combined with the previous generation to form a new population according to the survival procedures laid down. From here, 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.

2.3 Experiments

Daimler-Benz Research and Technology has conducted several experiments to investigate the applicability of evolutionary testing for testing the temporal behavior of different real world systems, and compared it to other procedures for the verification of temporal correctness. A Matlab-based toolbox that was developed at the Daimler-Benz Laboratories by Hartmut Pohlheim was applied for the evolutionary test. It provides a multitude of different evolutionary operators for selection, recombination, mutation, and reinsertion and also supports the use of subpopulations and competition [Pohlheim, 1996].

Wegener and Grochtmann [1998] compare evolutionary testing and random testing. Eight different test objects from varying application fields were tested. Two examples come from the field of computer graphics; another two come from the field of automotive electronics; one example is taken from railroad control technology, and one from the field of defense electronics. Some of these examples are safety-critical applications. A multiplication of matrices and a sorting algorithm were tested in addition. The examples vary from 12 LOC (lines of code) to 1511 LOC in their complexity, and from 8 input parameters to 5000 input parameters. For each sample test object the evolutionary test was applied twice, first, to find the best-case execution time, and then the worst-case. For random testing at least as many test runs have

been executed as for evolutionary testing. In all the experiments described, evolutionary testing obtained better results than random testing. More extreme execution times were found by means of evolutionary testing with a less or equal testing effort than for random testing. The disadvantage of a random method, that no step builds upon another, is avoided by using evolutionary testing. 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 gained from previous trials.

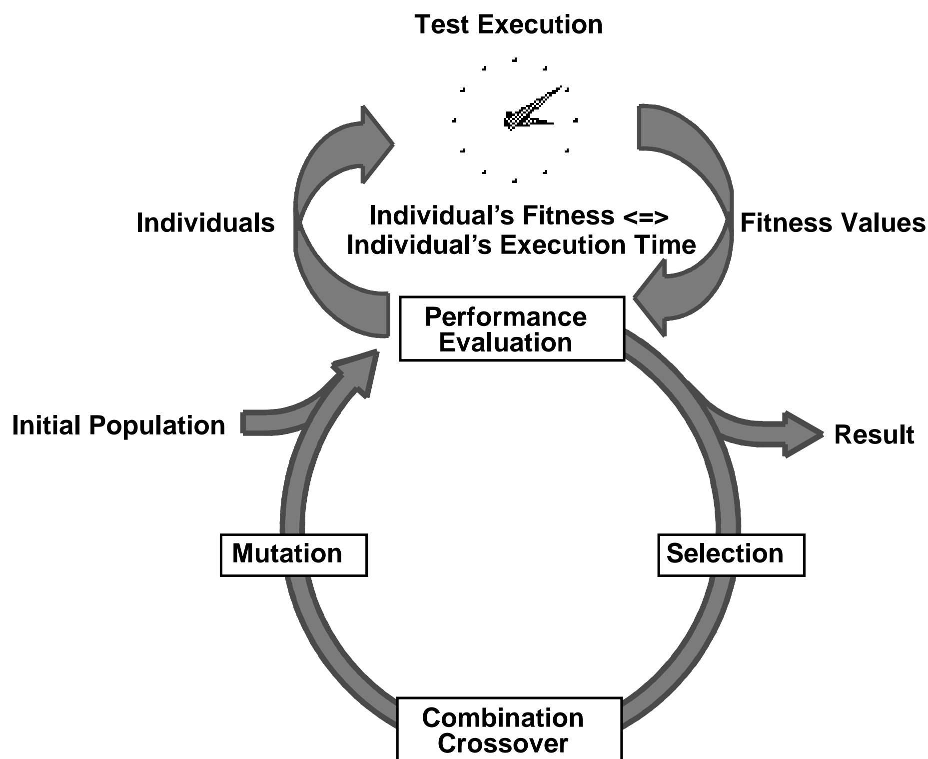


Figure 2. Evolutionary Testing of Temporal Behavior

The comparison of evolutionary testing and systematic testing show more ambiguous results [Grochtmann and Wegener, 1998]. On the one hand, an experiment comparing evolutionary testing with the results of systematic tests carried out by 19 different testers on two test objects from the field of computer graphics showed evolutionary testing to always be superior or equal to systematic testing. On the other hand, in an experiment testing a standard sorting routine which uses the bubblesort algorithm, evolutionary testing quickly approached the extreme execution times searched for but found the best-case and worst-case execution time only after several hundred more generations. In this case, systematic testing easily found the input with the extreme execution times: the already sorted list and the list sorted in reverse order. Thus, evolutionary testing seems to be better suited for complex test objects where the temporal behavior is difficult to assess by systematic testing. Further improvements are possible through the combination of evolutionary testing with systematic testing. 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 — they might take too long to find certain test relevant value combinations — can be compensated. Depending on the applied test method, evolutionary testing benefits from the tester's knowledge of the program function (functional testing) or the program structure (structural testing).

Mueller and Wegener [1998] compare evolutionary testing with timing predictions calculated by static analysis. Static timing analysis constitutes an analytical method for determining bounds on the best-case and worst-case execution times of applications. Static analysis simulates the timing behavior at a cycle level for hardware concepts such as caches and pipelines of a given processor. Timing estimates are calculated without knowledge of the input and without executing applications. For the comparison of evolutionary testing with static timing analysis both approaches are used in five experiments to determine the shortest and longest execution times of different programs. Two general-purpose algorithms and three industrial applications of Daimler-Benz were used as test objects. The experiments show that the methods of static analysis and evolutionary testing bound the actual execution times. For the worst-case execution time, the estimates of static analysis provide an upper bound while the measurements of evolutionary testing give a lower bound. Conversely, the static analysis estimates provide a lower bound for the best-case execution time while evolutionary testing constitutes an upper bound. In about half of the experiments, the actual shortest and longest execution times are bounded very precisely. In further cases, the two approaches vary by $\pm 10\%$. The accuracy of the results obtained by evolutionary testing is comparable to the accuracy of those of static analysis. If possible both approaches should be applied for the development of real-time systems: static analysis during the design phase and evolutionary testing to validate the temporal correctness of the implemented system.

In the described experiments, a previously determined number of generations was used as stopping criteria for the evolutionary test. In some cases this lead to the execution of more iterations for the test than would have been necessary for the determination of the best-case or worst-case execution time. In other experiments, however, the evolutionary test was broken off too early; a continuation of the test would have offered a good chance of finding more extreme execution times. For this reason, a more suitable termination criteria needs to be investigated.

3 Cluster Analysis

Cluster analysis for genetic algorithm output is a technique developed by Vössner and Braunstingl [1996] to provide more information about the population of individuals at a given generation of a genetic algorithm. Using cluster analysis on the population from the most recent generation produced by a genetic algorithm helps determine if the genetic algorithm should terminate. Additionally, selecting an appropriate generation from a genetic algorithm run and applying cluster analysis provides considerable information about the behavior of the fitness values over the search space.

3.1 Analyzing a Generation

To simplify cluster analysis for a generation, all parameter intervals are normalized to the range $0 \leq parameter \leq 1$, so that the parameter space, formerly a hyper rectangle, is now a hyper cube (with unit edge length and volume). In this normalized parameter space, the distance d_{ij} between two individuals *i* and *j* is taken with the Euclidean metric and calculated for all individuals. Doing this yields a symmetric distance matrix ($n \times n$) where $d_{ij} = d_{ji}$ for $i, j = 1, \ldots, n$.

Clusters of individuals are defined using the distance matrix. The distances of all individuals are compared with a particular threshold value d_{check} (also called the *check distance*). If the distance between two individuals is smaller than d_{check}, they lie in the same cluster.

Clusters identified by this algorithm may have arbitrary shapes, which is especially useful if the fitness function has an asymmetric gradient around a local optimum. Suppose the optimum is the highest point of a narrow ridge. Then the individuals will accumulate along this ridge, forming a longish, thin cluster.

The clustering algorithm is summarized in the following pseudo code, where a cluster consists of at least 2 individuals (solution points).

Given any two individuals *i* and *j*, if the distance between them $d_{ij} < d_{check}$, then they should be members of the same cluster. There are three cases:

1. If they do not belong to a cluster yet, they form a new one.
2. If only one of the two individuals already belongs to a cluster, the other one joins this cluster.
3. If both individuals are members of different clusters, these clusters are combined thus forming a single one.

3.2 Forming a Cluster Diagram

The clusters for a specific check distance are very dependent on that threshold value. Observing the clusters as the check distance changes provides complete information about the clustering of individuals in a generation. Decreasing the check distance by discrete factors, starting from the hypercube's space diagonal *d* (where exactly one cluster is found) down to a distance where no clusters exist, and performing the cluster analysis described above for each check distance forms a cluster diagram (see Figure 3).

The results of the different cluster analyses, ordered by decreasing check distance ($d/1$, $d/2$, $d/4$, …), are arranged in the cluster diagram, where each circle represents a cluster, and the connection from one cluster to the next outward cluster(s) shows how clusters split with decreasing check distance.

The diagram (Figure 3), generated by CLUSTER [Vössner and O'Sullivan 1998] shows the number of clusters, their sizes, history of origin, and the average fitness of the corresponding individuals. The numbers within the circles identify them, while the size of the circles is proportional to the number of individuals forming that cluster. The shading of a cluster indicates the average fitness of the individuals within it.

One of the shades is labeled with "avg" in the fitness scale at the bottom of the diagram. If a cluster number is underlined, the average fitness of its elements is greater or equal to the average fitness of all individuals. We call such a cluster "outstanding."

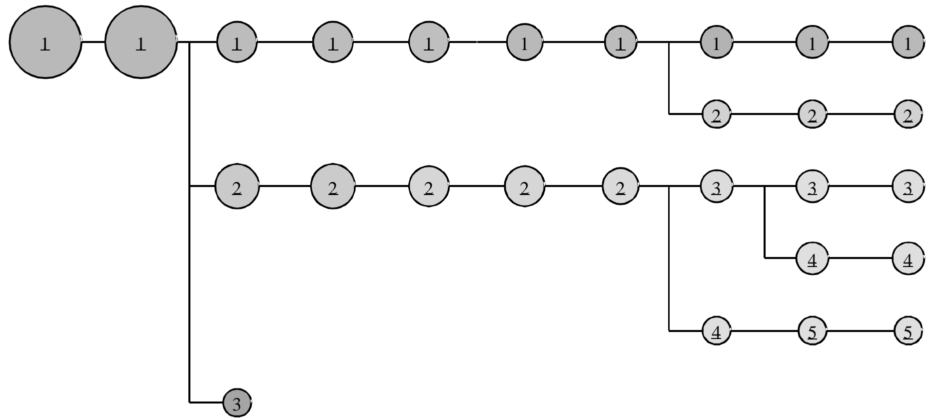
4 Measuring Convergence of Genetic Algorithms

In evolutionary testing, if an individual's fitness (execution time) violates the given timing constraints then the test is terminated. However, if no such individual is found, when should the test finish? The decision becomes that of deciding when to terminate a genetic algorithm. There are several criteria one may use to terminate a genetic algorithm, the following is a brief summary of these criteria.

Cluster-Tree for gen_0047.dat

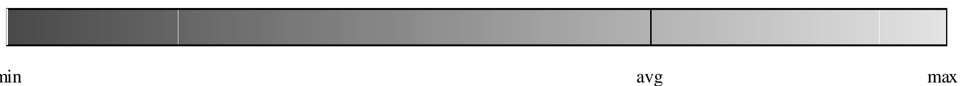
n = 70, d = 8.366600

d / 1 d / 2 d / 3 d / 4 d / 5 d / 6 d / 7 d / 8 d / 9 d / 10



Colortable - fitness

(min = 6099.000 - avg = 7812.880 - max = 8464.000)



CLUSTER 1.80 (29-Jan-1998) - (c) 1995-1998 GOL (Genetic Optimization Laboratory)

Figure 3. Example Cluster Diagram

4.1 Limiting number of generations/time

Limiting the number of generations or the period of time are the easiest termination criteria to use. However there is no guarantee that the genetic algorithm has converged at all after the set limit of generations or time.

4.2 Examining the fitness evolution

There are two different methods for using the fitness evolution to determine if a genetic algorithm has converged. One method examines the change in the maximum fitness. If there has been no improvement for a specified time the genetic algorithm terminates. However, when testing real-time systems' temporal behavior there would often be large areas of the search space over which the fitness value is constant [Mueller and Wegener, 1998]. In such regions the maximum fitness would remain unchanged for some time even though the algorithm has not converged.

The other method stops the genetic algorithm when a specified fitness value has been attained. In evolutionary testing this fitness value is the timing constraint specified for the unit under test. However, if the test object is temporally correct then this limit will never be attained and the algorithm will not terminate.

4.3 Looking at κ convergence

The κ convergence measure gives an estimate on the global state of spatial convergence of all individuals in a population (Figure 4). The ratio d_{ij}/d is the normalized distance between two individuals and lies in the interval [0, 1]. To construct a number proportional to the proximity of individuals we add all $\left(1-d_{ij}/d\right)$ between all M individuals.

$$\tilde{\kappa} = \sum_{i=1}^{M-1} \sum_{j=i+1}^{M} \left(1 - \frac{d_{ij}}{d}\right)$$

Complimentary distances d_{ij} and d_{ji} are only counted once. $\tilde{\kappa}$ achieves its maximum value when all individuals are at the same position. Their distance $d_{ij}/d \equiv 0$ and $\tilde{\kappa}$ computes to

$$\tilde{\kappa} = \tilde{\kappa}_{max} = \sum_{i=1}^{M-1} \sum_{j=i+1}^{M} 1 = \frac{M^2 - M}{2}$$

So $\tilde{\kappa}$ can finally be written as:

$$\tilde{\kappa} = \tilde{\kappa}_{max} - \frac{1}{d} \sum_{i=1}^{M-1} \sum_{j=i+1}^{M} d_{ij}$$

Eventually κ is found by normalizing $\tilde{\kappa}$ with $\tilde{\kappa}_{max}$:

$$\kappa = \frac{\tilde{\kappa}}{\tilde{\kappa}_{max}}$$

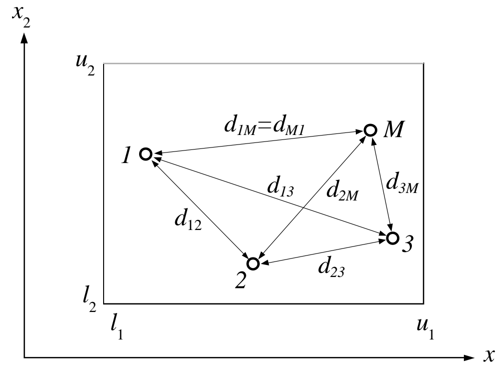


Figure 4. The κ convergence measure

If a population has converged to one point then κ is 1 and the algorithm has converged. However several global optima will complicate matters. This can lead to populations that are distributed around the different optima, so κ will not reach 1 even though the algorithm may have converged to these global optima.

4.4 Looking at ϕ convergence

The ϕ convergence measure gives an estimate of the fitness convergence value of all individuals in a population. Given the average fitness f_{avg} and maximum fitness f_{max} for the population,

$$\phi = \frac{f_{avg}}{f_{max}}.$$

If a population has converged to a single fitness value then ϕ is 1 and the algorithm has converged. However, this criterion faces the same problem as looking at the change in the maximum fitness, large regions in the search space that have the same fitness value may cause premature termination of the genetic algorithm.

4.5 Looking at the cluster analysis

When a genetic algorithm has converged to a global maximum, then the individuals in the population have clustered on one or more points with the same fitness value. By looking at the cluster analysis of the most recent generation, one can determine if this has taken place. If the clusters split suddenly from one cluster into several clusters that even exist at small check distances then the population has clustered around several points. If the fitness of these clusters is close to identical, then all these points have the same fitness value. Both of these conditions may be ascertained by examining the cluster diagram (see Figure 5) of the generation, or automatically by checking the size and fitness of the clusters.

This criterion may be less useful with small populations as there are not enough individuals to provide informative clusters. The larger the population the better the information provided by cluster analysis.

5 Application of Cluster Analysis and Evolutionary Testing

In principle, evolutionary testing should be ended as soon as the population has converged to one or more local and global optima since the chance of finding even better solutions is significantly reduced for converged populations. We have therefore examined the use of cluster analysis as termination criteria for evolutionary testing. Additionally, cluster analysis provides detailed information on local optima. For real-time systems this information could be useful to optimize the program code.

5.1 Convergence analysis

To examine the applicability of cluster analysis as termination criteria for evolutionary testing several experiments were performed. The test object is a complex algorithm from automotive electronics. It has 70 input parameters and 1511 lines of code. Seven evolutionary tests with different genetic algorithms were executed on the automotive electronics example in order to find the worst-case execution time. The execution times were measured in processor cycles. The experiments were performed with a population size of 15 individuals and were allowed to run for 400 generations.

Each of the runs returned a different solution. The longest execution times determined varied from 9459

processor cycles (run 2) to 12178 processor cycles (run 5). This indicates that either the solutions are local optima or the evolutionary tests were terminated too soon. The fitness evolution function for each of the runs is given in Figure 6.

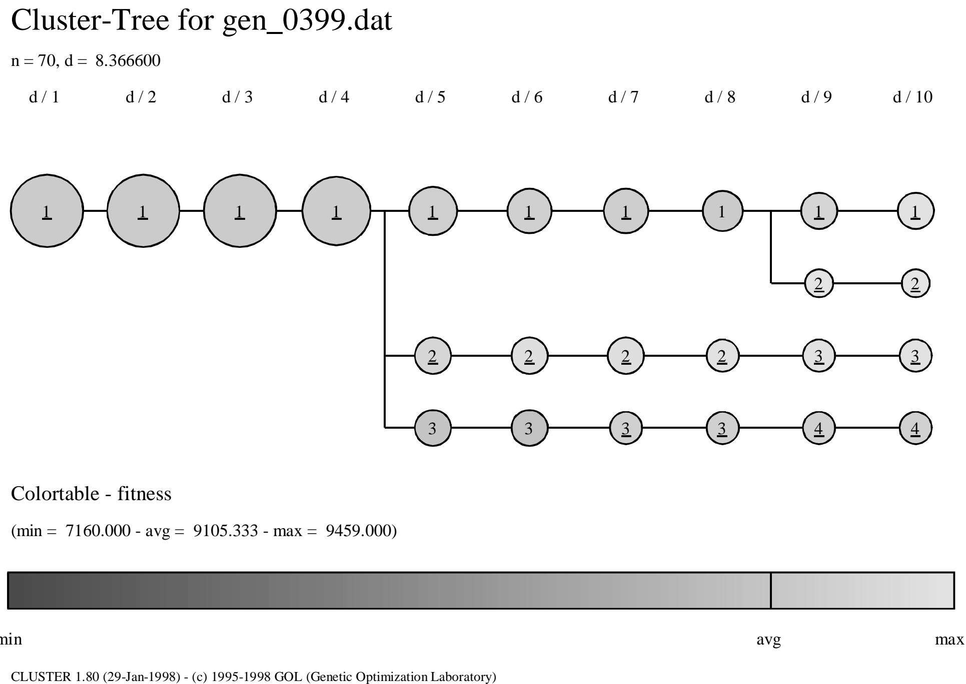


Figure 5. Cluster diagram for a convergent generation

To check on the convergence of each of the runs, cluster analysis was performed on the final population (generation 399) of each run. The cluster diagrams indicate that some of the tests are close to convergence, e.g. test run 1 and test run 3 (Figure 7), but others appeared to have been terminated too soon, e.g. test run 6 and test run 7 (Figure 8).

The cluster diagrams for test run 1 and test run 3 have a single cluster that splits into a few outstanding clusters. These clusters appear even at small check distances indicating these tests are nearly convergent. Accordingly, the chance of finding better solutions is poor when continuing these test runs. Both tests converged to local optima: the longest execution times found (11179 cycles and 10536 cycles) are far away from the worst-case execution time determined in all the experiments (12178 cycles).

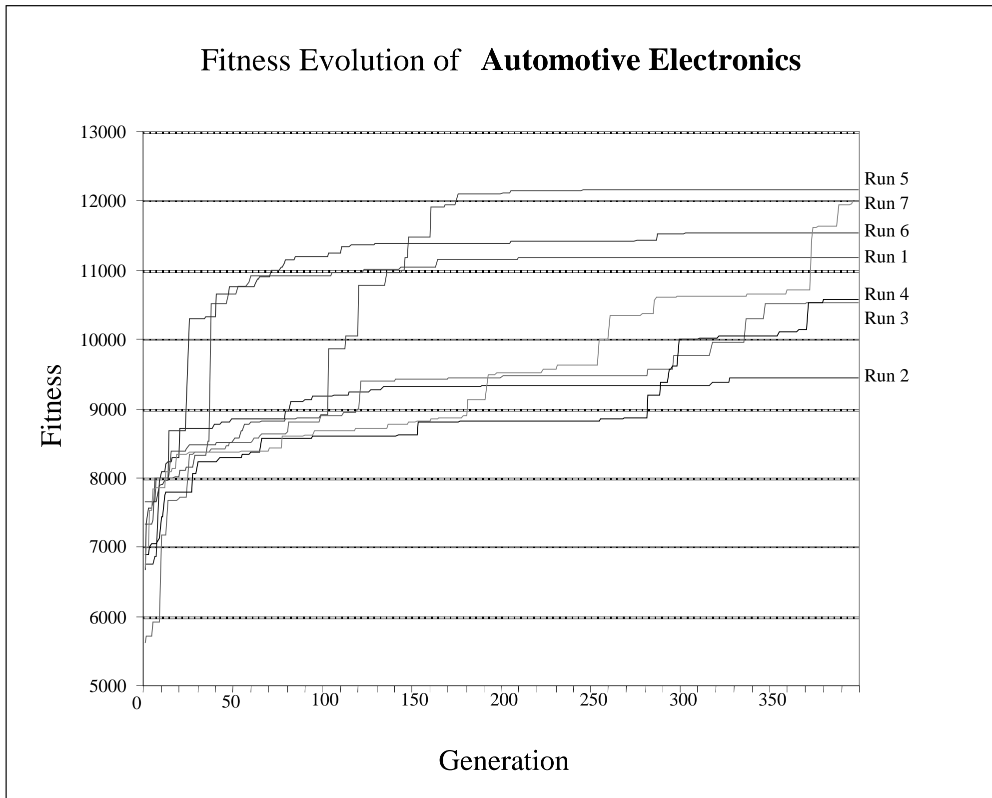


Figure 6. Fitness evolution for the *automotive electronics* tests

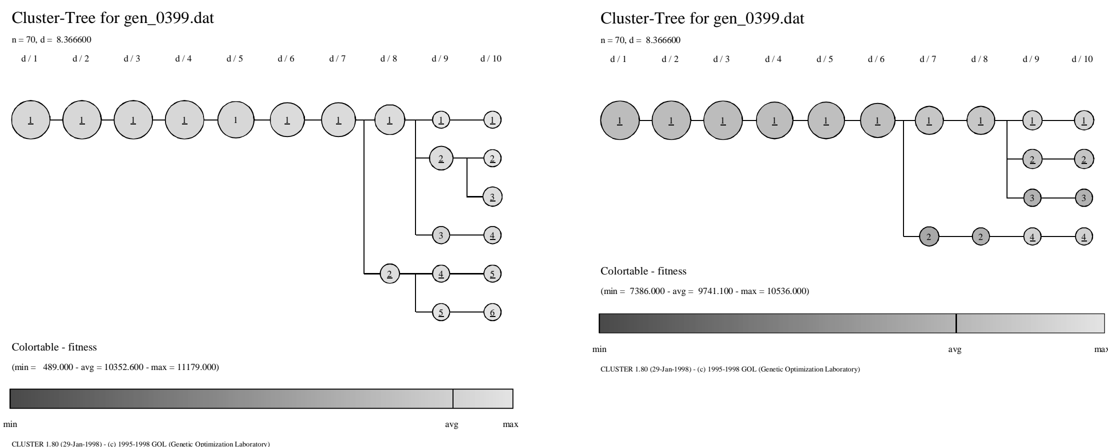


Figure 7. Cluster diagram for the final generation of test run 1 and test run 3.

The cluster diagrams for test run 6 and test run 7 show many clusters that are not outstanding and these clusters split as the check distance decreases so the tests are still some way from converging. The chance of hitting more extreme execution times is high. The tests are to be continued. From the cluster analysis

and the execution times found so far (11544 cycles and 11993 cycles) follows that especially test run 7 should be able to detect a new worst-case execution time for the automotive electronics system.

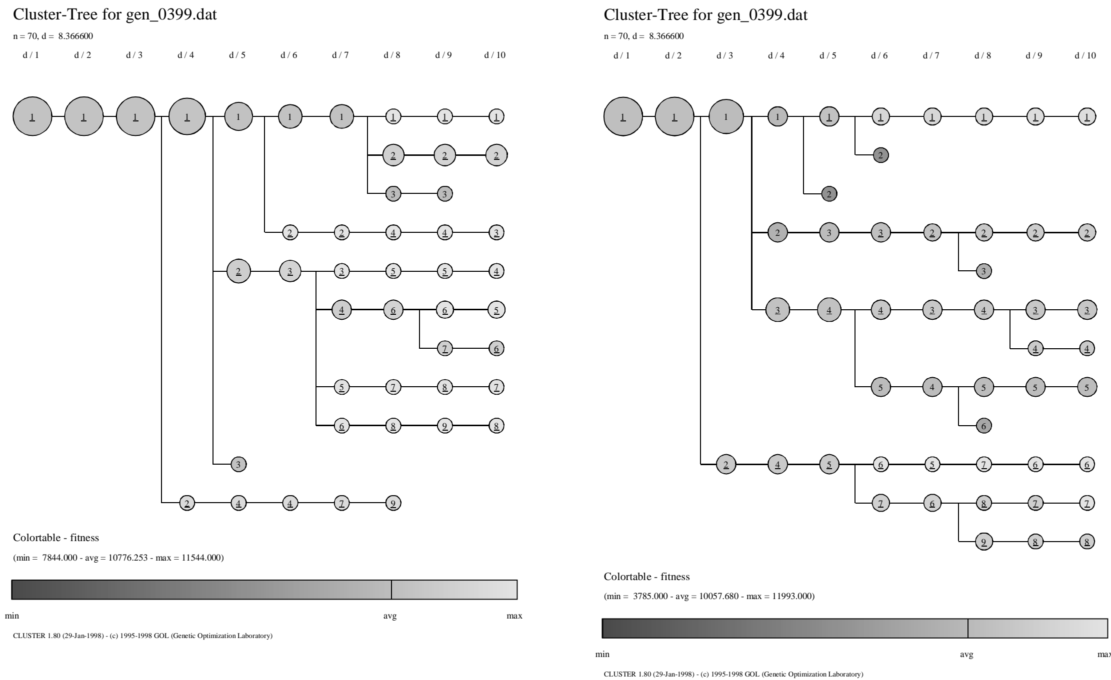


Figure 8. Cluster diagram for the final generation of test run 6 and test run 7.

To compare the results from cluster analysis with other termination criteria κ and ϕ were calculated for each of the tests. Table 1 gives the values for κ, calculated for the last generation. Figure 9 shows the evolution of ϕ over all generations for the test runs described in detail.

Run	κ
1	0.796512
2	0.718097
3	0.771224
4	0.719248
5	0.753917
6	0.734654
7	0.660362

Table 1. κ convergence measures for test runs

Table 1 shows κ has a value between 0.65 and 0.80 for all the experiments. In none of the experiments κ comes close to the convergence value of 1. For this reason, none of the tests would have been stopped on basis of κ. This hints at the existence of local optima which is confirmed for the test runs by the diagrams from cluster analysis (comp. Figure 7 and Figure 8).

The highest values for κ are calculated for the test runs 1, 3, and 5, the lowest value for test run 7. These results are comparable to those of cluster analysis. However, for the second test run κ has a

comparatively small value whereas cluster analysis determines a high convergence for this test. Conversely, for test run 6, κ calculates a high value whereas the cluster analysis diagram shows a population far away from convergence (Figure 8).

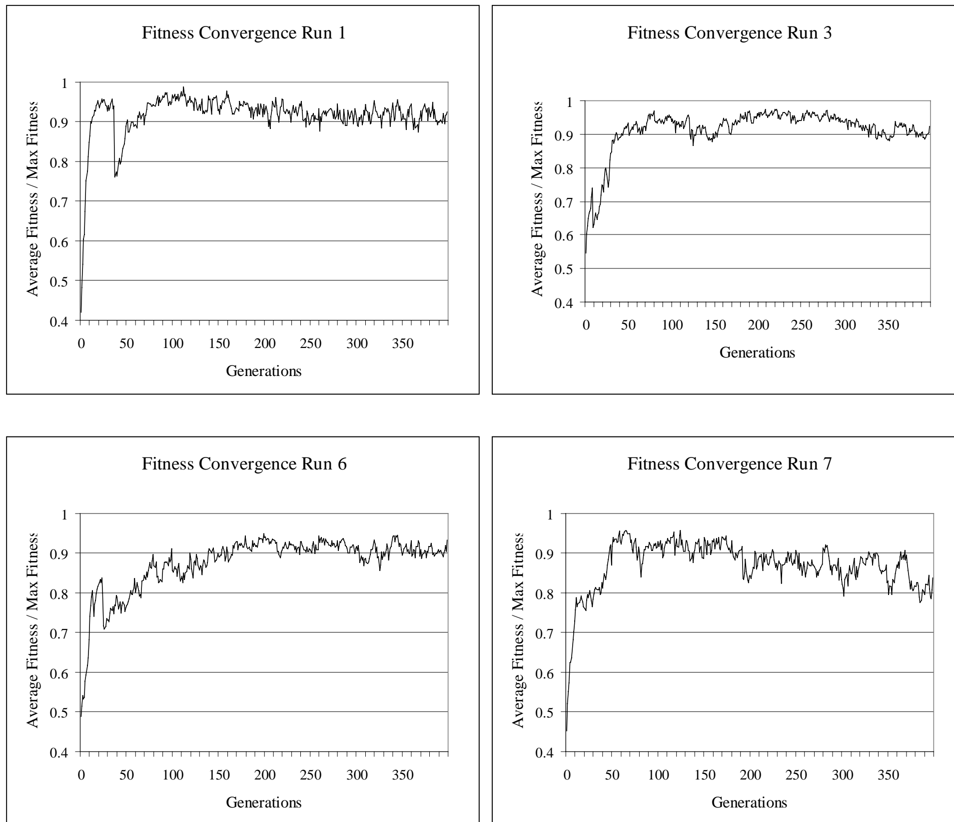


Figure 9. Evolution of the fitness ratio ϕ for the test runs 1, 3, 6, and 7

Analyzing the evolution of the fitness ratio ϕ for the experiments performed shows that for the test runs 1, 2, 3, and 6 ϕ has mostly high values in the range of 0.9 to 1.0. For the test runs 4, 5, and 7 values ranging from 0.8 to 0.9 are calculated for ϕ. These results show significant differences compared to those of cluster analysis, especially for test run 5 and test run 6.

A more detailed examination of Figure 9 indicates further problems when using ϕ as termination criteria, e.g. test run 7 achieves its highest fitness convergence in generation 70 approximately. At this stage the longest execution time found by this test is less than 9000 processor cycles. Stopping the test in accordance with ϕ after 70 generations would have prevented the finding of much longer execution times for test run 7 (11993 cycles in generation 399). The same is valid for most of the experiments. If ϕ had been used as termination criteria most of the tests would have been terminated too soon.

5.2 Detecting Local Optima

Using cluster analysis, finding local optima in an input domain is a relatively simple procedure, one that should take a similar amount of CPU time as finding the global optimum. Initially the appropriate generation must be selected from a previous test run. The clusters found by analyzing this generation are regions that are likely to contain local optima. By taking each of these clusters in turn, sampling an initial population from the cluster and running a new evolutionary test, the local optima may be refined from the cluster.

By taking note of the average fitness of the population throughout the initial evolutionary test, an appropriate generation for cluster analysis is selected. The average fitness of all generations of a test run yields function shapes similar to the ones shown in Figure 10.

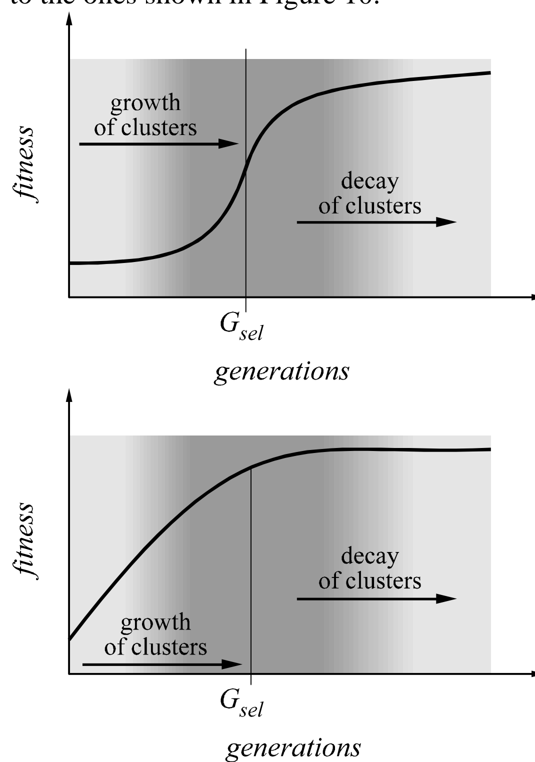


Figure 10. Selection of a generation for cluster analysis

When the individuals are distributed randomly, as in the initial population, no definitive clusters have formed. As the evolutionary test proceeds, individuals start moving towards areas of higher fitness, more clusters appear, and the average fitness increases rapidly. Then the individuals begin congregating into one or more optima. As the initial clusters start to disappear or split, the average fitness grows more slowly.

The following heuristic has turned out to be very useful for determining an appropriate generation to begin cluster analysis. We select the population G_{sel}, which comes closest to the point of inflection respectively, to the point of maximum curvature of the curves shown in Figure 10.

The original test run may terminate after few generations, only long enough for clusters to begin to form. The test run starting from each cluster may have a small to medium population size as the search is only in the neighborhood of the cluster. This run should also converge quickly as the population moves toward the local optimum. By keeping the test run with a large population short and using multiple test runs with small populations and quick convergence, first experiments have found local optima using approximately the same number of fitness evaluations as were required to find the global optimum. The local optima provide important information about the temporal behavior of the test object over the search space.

5.3 Discussion

The experiments performed demonstrate the value of incorporating cluster analysis into evolutionary testing. Cluster analysis is a useful termination criterion for evolutionary tests. Cluster diagrams especially provide an informative graphical method of checking for convergence. In comparison to other termination criteria cluster analysis seems to be the most powerful and promising termination criteria.

Limiting the number of generations or the time spent on the test is not a suitable termination criteria since there is no guarantee that the test has converged at all after the set limit. In our experiments some of the test runs were close to convergence after 400 generations, but using cluster analysis confirms that most of the tests were terminated too soon.

Examining the fitness evolution as stopping criteria is also not appropriate. In our experiments test run 6 shows long periods of stagnation although the population is far from convergence. Furthermore, test run 3 and 4 found even longer execution times after long periods of stagnation.

The ϕ convergence measure gives an estimate of the fitness convergence value of all individuals in a population. However, this criterion faces the same problem as looking at the change in the maximum fitness, large regions in the search space that have the same fitness value may cause premature termination of the test. This weakness is confirmed in our experiments. If the ϕ convergence measure had been used as termination criteria most of the tests would have been terminated too soon.

Like cluster analysis, the κ convergence measure gives an estimate on the spatial convergence of all individuals in a population. If a population has converged to one optimum in the search space κ is 1. However, κ is insufficient for combination with evolutionary testing since the temporal behavior of most real-time systems contains several optima, so κ will not reach 1 even though the evolutionary test may have converged. Accordingly, in our experiments the values for κ were in the range 0.65 to 0.8 only.

In addition to test termination, cluster analysis allows local optima to be located quickly. The local optima in an evolutionary test give important information about the behavior of the test algorithm over the input domain. However, there are some caveats when using cluster analysis. Cluster analysis is only as extensive as the evolutionary test it is used with. If the test locates a subset of the local optima, then clusters will only have formed in these areas. Starting new tests from within these clusters will not find all the local optima. Using a large population size for a small to medium number of generations should alleviate this concern, the population will spread throughout the entire search space and the clusters will be meaningful, yet not too restrictive, giving good starting points for local optima searches.

6 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 evolutionary testing has been successfully applied to search the worst-case and best-case execution times of real-time programs in order to check whether they violate their specified timing constraints.

Further improvements are possible through the combination of evolutionary testing with cluster analysis. First of all, cluster analysis aids in forming a reliable termination criterion. The clusters provide detailed information about the convergence state of the evolutionary test. A test which has converged to one or more optima should be terminated because the probability of finding even better solutions is small. Cluster analysis also provides additional information about local optima in order to detect several performance leaks within one test run. It gives information about the temporal structure of the entire search space. This is a good basis for optimizing the program code of real-time systems.

Evolutionary testing shows 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 the underlying genetic algorithms. Complementary to cluster analysis the degree of coverage (e.g. branch coverage) achieved during evolutionary testing and the observation of the program paths executed could be an interesting aspect for deciding when to stop the test. Additionally, further studies should focus on the question how cluster analysis information can be used to react to stagnations with appropriate changes of the search strategy.

References

Boden, E.B., and Martino, G.F. (1996). *Testing Software Using Order-Based Genetic Algorithms*. Proceedings of Genetic Programming ’96, pp. 461 - 466, July 1996, Stanford University, USA.

Goldberg, D.E. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, USA.

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

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.

Grochtmann, M., and Wegener, J. (1998). *Evolutionary Testing of Temporal Correctness*. Proceedings of Quality Week Europe ’98, 9 - 13 November 1998, Brussels, Belgium.

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

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

Jones, B.F., Eyres, D.E., and Sthamer, H.-H. (1998). *A Strategy for using Genetic Algorithms to Automate Branch and Fault-based Testing.* The Computer Journal. Vol. 41, No. 2, pp. 98 - 107.

Jones, B.F., Sthamer, H.-H., Yang, X., Eyres, D.E. (1995). *The Automatic Generation of Software Test Data Sets using Adaptive Search Techniques.* Proceedings of Software Quality Management ’95, pp. 435 - 444 Seville, Spain.

Mueller, F., and Wegener, J. (1998). *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, 3 - 5 June 1998, Denver, USA.

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

Roper, M. (1997). *Computer Aided Software Testing using Genetic Algorithms*. Proceedings of Quality Week ’97, 27 - 30 May 1997, San Francisco, USA.

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.

Schütz, W. (1993). *The Testability of Distributed Real-Time Systems.* Kluwer Academic Publishers, Boston, USA.

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.

Vössner, S. and Braunstingl, R. (1996). *G.O.A.L (Genetic Optimization Algorithm).* Genetic Optimization Lab, Technical University Graz, Austria.

Vössner, S. and O’Sullivan, M. (1998). *CLUSTER (Cluster Analysis Package for Genetic Algorithms).* Genetic Optimization Lab, Stanford University, CA, USA.

Wegener, J., Grochtmann, M., and Jones, B. (1997). *Testing Temporal Correctness of Real-Time Systems by Means of Genetic Algorithms*. Proceedings of Quality Week ’97, 27 - 30 May 1997, San Francisco, USA.

Wegener, J., and Grochtmann, M. (1998). *Verifying Timing Constraints of Real-Time Systems by Means of Evolutionary Testing*. To appear in Real-Time Systems.