Automatic Test Data Generation for Structural Testing of Embedded Software Systems by Evolutionary Testing

Joachim Wegener, Kerstin Buhr, Hartmut Pohlheim
DaimlerChrysler AG, Research and Technology
Alt-Moabit 96a, D-10559 Berlin, Germany, +49 30 39982 232
{Joachim.Wegener, Kerstin.Buhr, Hartmut.Pohlheim}@DaimlerChrysler.com

Abstract

Testing is the most important analytic quality assurance measure for software. The systematic design of test cases is crucial for test quality. Structure-oriented test methods, which define test cases on the basis of the internal program structures, are widely used.
Evolutionary testing is a promising approach for the automation of structural test case design which searches test data that fulfil given structural test criteria by means of evolutionary computation.
In this paper we present our evolutionary test environment, which performs fully automatic test data generation for most structural test methods. We shall report on the results gained from the testing of real-world software modules. For most modules we reached full coverage for the structural test criteria.

1 INTRODUCTION

A great number of today's products is based on the deployment of embedded systems. In industrial applications embedded systems are predominantly used for controlling and monitoring technical processes. There are examples in nearly all industrial areas, for example in aerospace technology, railway and motor vehicle technology, process and automation technology, communication technology, process and power engineering, as well as in defense electronics. Nearly 90% of all electronic components produced today are used in embedded systems.

In order to achieve high quality in the development of embedded systems, central importance is attributed to analytical quality assurance. In practice, the most important analytical quality assurance measure is dynamic testing. Thorough testing of the systems developed is essential for product quality. The aim of the test is to detect errors in the system under test, and, if no errors are found during comprehensive testing, to convey confidence in the correct functioning of the system. This is the only procedure which allows the testing of dynamical system behavior in a real application environment.

The most significant weakness of the test is that the postulated functioning of the tested system can, in principle, only be verified for those input situations selected as test data. Testing can only show the existence and not the non-existence of errors. Therefore, the correctness proof can only be produced by a complete test. In practice, a complete test, with the exception of a few trivial cases, is not executable because of the enormous amount of possible input situations. Thus, the test is a sampling procedure. Accordingly, a task which is essential to testing is the selection of an appropriate sample containing the most error-sensitive test data.

Among the different test activities (test case design, test execution, monitoring, test evaluation, test planning, test organization, and test documentation – see Fig. 1) test case design is of essential importance.

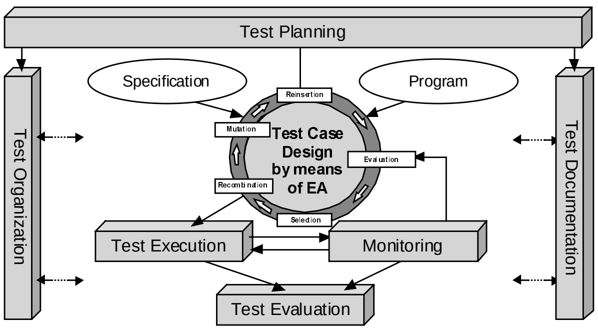

Fig. 1 Structure and interaction of test activities including test case design by means of evolutionary algorithms

For most test objectives an automation of test case design is difficult to achieve. Thus, test case design usually has to be performed manually. Manual test case design, however, is time-intensive and susceptible to errors. The quality of the testing is heavily dependent on the performance of the single tester.

To increase the effectiveness and efficiency of the test, and thus to reduce the overall development costs for software-based systems, we require a test which is systematic and extensively automatable. For this reason, DaimlerChrysler Research works in the area of *evolutionary testing* [19]. The aim of the work is to increase the quality of the tests and to achieve substantial cost savings in system development by means of a high degree of automation of test case design.

Evolutionary testing is a promising approach for fully automating test case design for various test aims. For instance, evolutionary tests can be used to systemize and automate the testing of non-functional properties and to generate test cases for conventional test methods. In this paper we shall concentrate on structural testing.

The only prerequisites for the application of evolutionary testing are an executable test object and its interface specification. For the automation of structural testing the source code of the test object must be available to enable its instrumentation.

In Section 2 we give a short overview of evolutionary testing. Section 3 describes the different aspects of structural testing. Our evolutionary test environment is presented in Section 4. A large number of real-world software modules have already been tested. The results are presented in Section 5. Our concluding remarks are set out in Section 6 along with our outlook for future research.

2 EVOLUTIONARY TESTING

Evolutionary testing is characterized by the use of meta-heuristic search techniques for test case generation (see Fig. 1). The test aim considered is transformed into an optimization problem. The input domain of the test object forms the search space in which one searches for test data that fulfils the respective test aim. Additionally, a numeric representation of the test aim is necessary. This numeric representation is used to define objective functions suitable for the evaluation of the generated test data. Depending on which test aim is pursued, different objective functions emerge for test data evaluation. Section 4 describes objective functions for structural testing in detail.

Due to the non-linearity of software (if-statements, loops etc.) the conversion of test problems to optimization tasks mostly results in complex, discontinuous, and non-linear search spaces. Neighborhood search methods like hill climbing are not suitable in such cases. Therefore, meta-heuristic search methods are employed, e.g. evolutionary algorithms, simulated annealing, or taboo search. In our work, evolutionary algorithms are used to generate test data because their robustness and suitability for the solution of different test tasks has already been proven in previous work, e.g. [14], [6], [19].

As we assume the reader to be familiar with evolutionary algorithms we shall only describe the interaction of the evolutionary algorithm with the other testing activities in this paper. Please refer to [18] and [17] for a longer description of evolutionary algorithms in the context of evolutionary testing.

Figure 2 presents the structure of evolutionary testing from the point of view of the evolutionary algorithm. The interaction with the other testing activities occurs during the evaluation of the individuals.

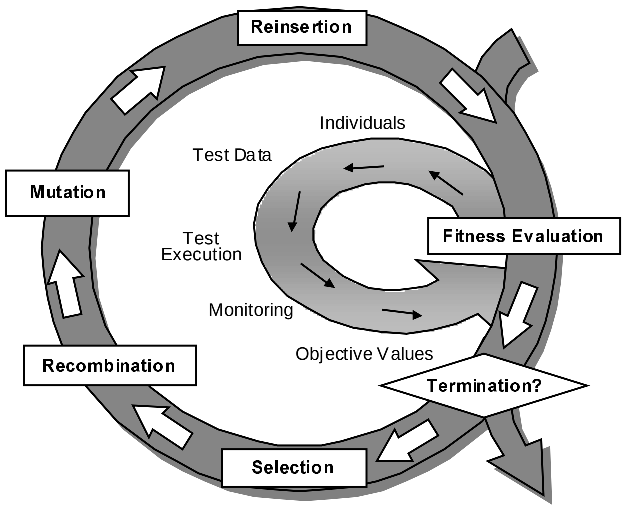

Fig. 2 Structure of Evolutionary Testing

Each individual within the population represents a test datum with which the test object is executed. For each test datum the execution is monitored and the objective value is determined for the corresponding individual. Next, population members are selected with regard to their fitness and subjected to recombination and mutation to generate new offspring. It is important to ensure that the test data generated are in the input domain of the test object. Offspring individuals are then evaluated by executing the test object with the corresponding test data. A new population is formed by combining offspring and parent individuals, according to the survival procedures defined. From now on, this process repeats itself until the test objective is fulfilled or another given termination criterion is reached.

3 STRUCTURAL TESTING

Structural testing is widespread in industrial practice and stipulated in many software-development standards, e.g. [13], [5], and [3]. The execution of all statements (statement coverage), all branches (branch coverage), or all conditions with the logical values True and False (condition coverage) are common test aims. Structural test methods are usually applied to unit tests. There are no enforced structure test criteria for integration tests or system tests.

The aim of applying evolutionary testing to structural testing is the automatic generation of a quantity of test data, which leads to the best possible coverage of the respective structural test criterion. In the case of statement testing, the goal of the test is to execute each program statement at least once. In the case of branch testing the empty branches also have to be executed. The test goals are based on the assumption that a

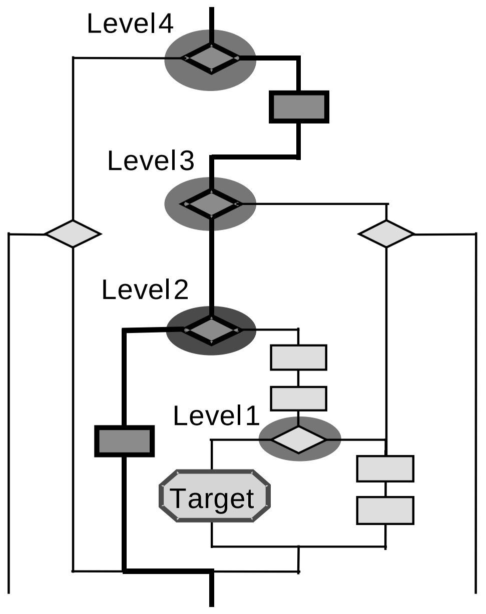

Figure 3. Approximation level calculation (objective function for structural testing)

test, which does not include each statement and all branches (of the system under test being executed) at least once, does not present a thorough check of the test object. Therefore, the overall goal of the test case design is to define a set of test data which guarantees that each statement or each branch is executed.

Whereas all previous work has concentrated on selected structural test criteria (statement-, branch-, condition and path test), our test environment has been developed to support all common control-flow and data-flow oriented test methods.

In order to search for the test data set the test is divided into partial aims. Each partial aim represents a program structure that requires execution to achieve full coverage, for example a certain statement or branch. For each partial aim, an individual objective function is formulated and a separate optimization is undertaken to search for a test datum executing the partial aim.

In order to direct the search toward program structures not covered, the objective function computes a distance for each individual that indicates how far away it is from executing the desired program structure. Individuals which are closer to the execution of the desired program structure will be selected as parents and combined to produce offspring individuals.

The objective functions of the partial aims consist of two components – the approximation level and the local distance calculation.

3.1 APPROXIMATION LEVEL CALCULATION

The approximation level supplies a figure for an individual that gives the number of branching nodes lying between program structures covered by the individual and the desired program structure. For this computation, only those branching nodes are taken into account that contain an outgoing edge that results in a miss of the desired program structure.

An example is given in Figure 3. The program graph contains four branching nodes which could result in a miss of the target node (representing the desired statement or branch). Each node is assigned the corresponding approximation level (level 4 to level 1). An individual that branches away from the target node at the first branching node attains a lower approximation level (level 4) than an individual which reaches level 3 etc. The figure shows the execution of an individual which misses the target node in the branching node with the approximation level 2. The individual passes the branching nodes in the levels 4 and 3 as desired but misses the target node at level 2.

3.2 LOCAL DISTANCE CALCULATION

The calculation of the local distance is performed in order to distinguish different individuals executing the same program path. For this, a distance to the execution of the other program path is calculated for the individual by means of the branching conditions in the branching node in which the target node is missed. Figure 4 illustrates this calculation.

For example, if a branching condition x==y needs to be evaluated as True to reach the target node, then the objective function may be defined as |x-y|. If an individual obtains the local distance 0, a test datum is found which fulfils the branching condition: x and y have the same value.

If a branching node contains multiple conditions the local distance is a combination of the local distances of each condition. For a node of the type a $\vee$ b the local distance of an

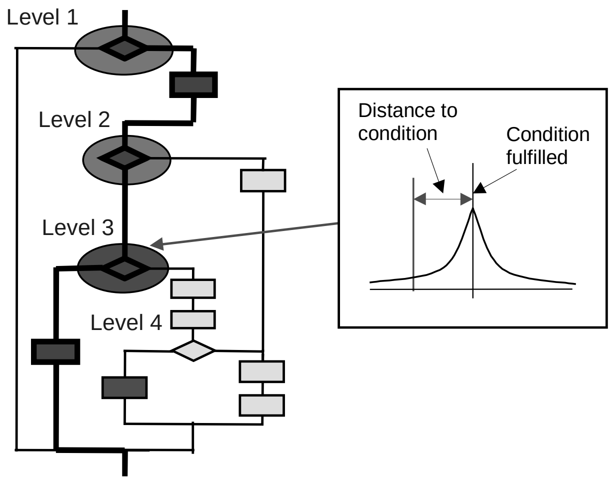

Figure 4. Local distance calculation (objective function for structural testing)

individual is obtained from the minimum value for each single predicate a and b. In the case of a $\wedge$ b the local distance of an individual is the result of the sum of the distances determined for each single predicate.

3.3 OBJECTIVE FUNCTION

The overall objective function value of an individual with respect to a certain partial aim is defined as the sum of its approximation level and its normalized local distance:

$F(pa, i) = AL(pa, i) + (1 - LD(N(pa, AL(pa, i)), i),$

- F(pa, i): objective value of individual i for the partial aim pa,
- AL(pa, i): highest approximation level of the individual i for the partial aim pa,
- N(pa, al): branching node with the highest approximation level for the partial aim pa,
- LD(n, i): normalized local distance of the individual i in branching node n.

An individual with an objective value of 0 leads to the passing of the desired program structure. This provides a natural termination criteria for the optimization of this partial aim.

Even though the evolutionary test works up only one partial aim after the other, the execution of a test datum usually leads to passing several partial aims. Thus, the test soon focuses on those program structures which are difficult to reach. After having worked up all partial aims, a minimal amount of test data is returned to the tester. This test data set leads to an execution of all reached partial aims.

4 EVOLUTIONARY TEST ENVIRONMENT

In order to automate test case design for different structural testing methods with evolutionary tests we have developed a tool environment which consists of six components:

- parser for the analysis of test objects,
- graphical user interface for the specification of the input domain of the test objects,
- instrumenter which captures program structures executed by the generated test data,
- test driver generator which generates a test bed running the test object with the generated test data,
- test control which includes the identification and administration of the partial aims for the test and which guarantees an efficient test by defining a processing order and storage of initial values for the partial aims,
- toolbox of evolutionary algorithms to generate the test data.

Fig. 5 presents the structure of the evolutionary test environment and shows the information exchange between these tools. The parser, interface specification, instrumenter and test driver generator constitute the test preparation. During

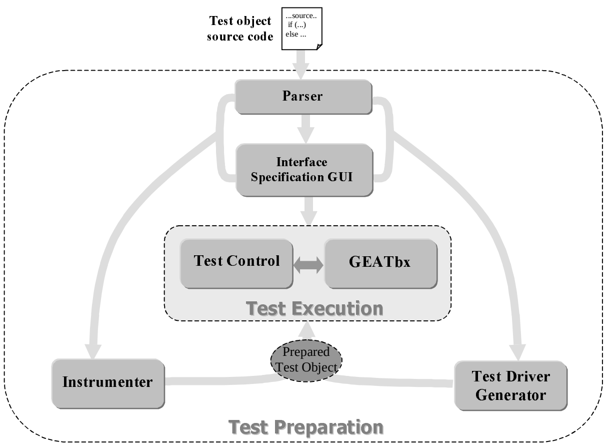

Figure 5. Components of the evolutionary test environment

test execution the test control and the evolutionary algorithm toolbox are employed.

4.1 PARSER

The parser identifies the functions in the source files which form the possible test objects. It determines all necessary structural information on the test objects. Control-flow and data-flow analyses are carried out for every test object. These analyses determine the interface, the control-flow graph, the contained branching conditions with their atomic predicates, as well as semantic information on the used data structures, e.g. the organization of user-defined data types.

4.2 GRAPHICAL USER INTERFACE FOR INTERFACE SPECIFICATION

To ensure efficient test data generation and to avoid the generation of inadmissible test data from the beginning, the tester may have to define the test object interface determined by the parser more precisely. For this, the developed tool environment provides a graphical user interface that displays the test objects and their interfaces as they have been determined by the parser.

The tester can limit the value ranges for the input parameters and enter logical dependencies between different input parameters. These will then be considered during test data generation. It is also possible to enter initial values for single or for all input parameters. As a result, test data of a previous test run or data of an already existing functional test, as well as specific value combinations for single input parameters, can be used as a starting point for test data generation (seeding).

4.3 INSTRUMENTER

The third component in the tool environment is the instrumenter that enables test run monitoring. In order to eliminate influences on program behavior the instrumentation has to take place in the branching conditions of the program. The instrumentation of the branching conditions is always the same, independent of the selected structural test criterion.

The atomic predicates in the branching nodes of the test object are instrumented to measure the distances individuals are away from fulfilling the branching conditions (see Section 3.2). The instrumentation also provides information on the statements and program branches executed by an individual.

4.4 TEST DRIVER GENERATOR

The test driver generator generates a test bed that calls the test object with the generated individuals and returns the monitoring results provided by the execution of the instrumented test object to the test control. When the test object is called by the test driver, the individuals are mapped onto the interface of the test object. It is important that user specifications for the test object interface are taken into account. Individuals that do not represent a valid input are extracted and assigned a low fitness value.

4.5 TEST CONTROL

The most complex component of the evolutionary test environment is the test control. It is responsible for several difficult tasks: the management of the partial aims with their processing status, the collection of suitable initial values for the optimization of partial aims, and the recording of test data fulfilling partial aims.

The test control identifies partial aims for the selected structural test criterion. Partial aims are determined on the basis of the control–flow graph provided by the parser. The test control manages the determined partial aims and regulates the test progress. One after the other, the different partial aims are selected in order to search for test data with the evolutionary test. Independent optimizations are performed for every partial aim.

Although only one partial aim is considered for the optimization at a time, all individuals generated are evaluated with regard to all unachieved partial aims. Thus partial aims reached by chance are identified, and individuals with good objective values for one or more partial aims are noted and stored. Subsequent testing of these partial aims then uses the stored individuals as initial values (also compare [9]). This method is called seeding. It enhances the efficiency of the test because the optimization does not start with an entirely randomly generated set of individuals.

In order to calculate the objective values for the individuals the test control determines the program paths executed by every individual, on the basis of the data attained by test monitoring and the test object's control-flow graph. Objective values are then evaluated by applying the objective functions described in Section 3, that take into account the local distance measurement for the branching conditions as well as the approximation level.

The processing sequence for the partial aims that have not yet been attained is guided by the test control depending on the availability of suitable initial values. The partial aim for which the individuals with the best objective values are available is selected as the next one for the test. This ensures that the test quickly achieves a high coverage because partial aims which are difficult to execute or infeasible do not slow down the overall testing process. When no initial values are available, or several equally good initial values for different partial aims exist, then a breadth-first search is carried out. If the search fails to find a test datum for a partial aim it is marked as already processed and not fulfilled. During the remaining optimization process it is possible to reset this status if an individual with a better objective value for this partial aim is found accidentally. The partial aim is then targeted again for an additional test employing the attained values for initialization.

Once all partial aims have been processed the test is finished. The test data for the separate partial aims are then compiled and displayed with the obtained coverage. On this basis, the tester is able to check whether program structures that were not covered are infeasible, or whether the evolutionary test was not able to generate suitable test data.

In addition, the test control offers a simple application programming interface (API) to export test data found, and actual values for the output parameters of the test object, in order to support automatic test evaluation on the basis of test oracles. Moreover, it provides test and monitoring information for the visualization of the test progress.

4.6 GENETIC AND EVOLUTIONARY ALGORITHMS TOOLBOX GEATBX

We have applied the *Genetic and Evolutionary Algorithm Toolbox for Use with Matlab* (*GEATbx*) [10]. This is a very powerful tool that supports real and integer number representation of individuals as well as binary coding. Almost any hybrid form of evolutionary algorithm can be implemented, including genetic algorithms and evolution strategies. The toolbox offers a large number of different operators for the components of evolutionary algorithms, and also enables the application of sub-populations, migration and competition between sub-populations, and possesses extensive visualization functions for displaying the optimization state and progress. It is possible to specify admissible value domains for the parameters of an individual. The toolbox automatically ensures that these value domains are observed during the generation of individuals. Thus, the test driver only needs to check for dependencies between the single variables of an individual.

5 APPLICATION OF EVOLUTIONARY TESTING

Our tool environment has already been applied in experiments with typical real-world examples. Currently, our work concentrates on the automatic generation of test data for statement and branch tests, which has yielded excellent results. For all test objects a complete or very high coverage was achieved by the evolutionary test.

5.1 TEST OBJECTS

Table 1 presents a selection of examined test objects. To assess and compare the complexity of the software modules we report a number of software metrics. These figures are taken from a larger study examining the complexity of more than 40 different software modules [2].

The basic metric is lines of code. The cyclomatic complexity gives information on the test object's control flow. The nesting complexity assesses the nesting level and can indicate the difficulty in reaching a partial aim with respect to the control-flow. Myer's interval shows the complexity of certain branching conditions.

Asin() calculates the arcsin or arccos for the passed argument (1 double) and is a typical C library function.
Prototype: double asin(double arg);

Atof() is another typical C library function. It converts strings to the corresponding floating point value. *Atof()* contains several evaluations which check the input string for its validity. In the experiments the maximum string length was set to 10 characters in ASCII coding. Accordingly, the size of the search space is 255^{10}. For this test object each test datum (individual of the evolutionary algorithm) consists of 10 integer variables, each with a possible value of 1 to 255.
Prototype: double atof(char InStr[10]);

The *classifTria()* function is an implementation of the classic triangle classifier example used in a large number of testing papers. It is used in two different data type versions. The input domain is given either by three floating point values or by three integer values.
Prototype: void classifTria(double a, double b, double c);

In *powi()* a passed floating point value is raised to a passed integer value. The input consists of 1 double and 1 integer.
Prototype: double powi(double x, int nn);

Incbet() is a larger test function. It calculates the incomplete beta-integral of the passed argument (3 doubles).
Prototype: double incbet(double aa, double bb, double xx);

In the experiments the double values were bounded in [-10^6, 10^6].

5.2 EVOLUTIONARY TESTING SETUP

Evolutionary testing was carried out using an evolutionary algorithm with the following configuration:

- linear ranking with a selective pressure of 1.7,
- selection by stochastic universal sampling,
- generation gap of 0.9,
- discrete recombination with a recombination rate of 1,
- real or integer valued mutation employing different mutation range for each subpopulation in the range [0.1, 0.01, ..., 10^{-6}] and a mutation rate of (1/number of variables),
- regional population model dividing the population into subpopulations,
- migration between subpopulations every 20 generations in a complete net structure (5% migration rate),
- competition between subpopulations every 5 generations (division pressure of 3),
- maximal number of generation of 200.

The sizing of the subpopulations depends on the complexity of the software module under test. We employed 9 subpopulations with 100 individuals each. This number is relatively large and therefore more appropriate for the complex software modules (*powi()* and *incbet()*). In order to compare the results more easily we used this number for all the modules in all the experiments.

Structural testing exhibits a natural termination criteria. As soon as a partial aim is reached the corresponding optimization process can terminate. Thus, an upper bound for the number of generations is only defined if a partial aim can not be reached.

There is one straightforward mechanism for the dynamic adaptation of evolutionary testing in the context of our work which has already been successfully employed: the use of

Table 1 Metrics and number of partial test aims of the used test objects

metrics/modules	**asin**	**atof**	**classifTria**	**powi**	**incbet**
lines of code	13	36	41	51	159
cyclomatic complexity	4	16	14	15	23
Myer's interval	0	27	7	2	3
nesting complexity	4	32	17	19	43
no. of statement cover aims	10	40	30	36	58
no. of branch cover aims	12	56	42	49	79

multiple strategies and competing subpopulations. The use of multiple strategies (different mutation range for each subpopulation) leads to different search strategies: from a globally oriented search when employing a large mutation range to a very fine search when employing a small mutation range. Additionally, depending on the test process, the most successful strategies will be assigned more resources. This leads to an efficient distribution of resources during the whole optimization and a more robust search. For a longer discussion of competing subpopulations see [11].

5.3 COMPARISON OF EVOLUTIONARY AND RANDOM TESTING

We compare the results of evolutionary testing (ET) to those of random testing (RT) for all test objects. The availability of other means of comparison is very limited. One could also compare the results with those of an expert with good knowledge of the modules under test. However, this would involve a great deal of effort which would not be justifiable for real-world problems. We have carried out this kind of comparison for several modules in order to test temporal behavior [12]. In these cases RT performed nearly as well as the expert, whereas ET always proved itself to be better or at least as good as the expert.

The results of the experiments are presented in Table 2. The number of individuals generated for random testing was equivalent to that for evolutionary testing. Each experiment was repeated 10 times. The mean values for the respective results are presented.

The evolutionary test achieved full statement and branch coverage for the first 3 software modules in a very short time. Random testing was unable to reach full coverage for any of the software modules. The coverage values are much lower in general.

For the more complex software modules *powi()* and *incbet()* evolutionary testing reached high coverage values. However, full coverage (100%) was not reached. We are still investigating if these coverage values are the highest possible values (because of infeasible statements and branches). It is thus quite possible that for these test objects, the maximal possible coverage has been reached.

When we compare the results of evolutionary testing and random testing for these two modules the advantage of evolutionary testing is much more apparent. Especially for *incbet()*, the coverage of random testing is substantially lower than that of evolutionary testing.

This suggests, that evolutionary testing is currently the only sensible procedure of structural testing of large and complex software modules.

6 CONCLUDING REMARKS

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

The aim of the work presented in this paper is the automatic generation of test data for structural tests. For this, a tool environment has been developed that applies evolutionary testing to C programs. Test data are generated by means of evolutionary algorithms.

With evolutionary testing a new test method for testing embedded systems is provided, which enables the complete automation of test case design for various test objectives. The idea of evolutionary testing is to search for relevant test cases in the input domain of the system under test with the help of evolutionary algorithms. As described in this paper, evolutionary testing enables the complete automation for structural test case design.

Due to the full automation of the evolutionary testing, the system could be tested with a large number of different input situations. In most cases, more than several thousand test data sets are generated and executed within only a few minutes. The prerequisites for the application of evolutionary

Table 2 Results of statement and branch coverage for evolutionary testing (ET) and random testing (RT)

	asin	**atof**	**classifTria**	**powi**	**incbet**
statement cover	ET / RT	ET / RT	ET / RT	ET / RT	ET / RT
coverage [%]	100 / 50	100 / 61.5	100 / 13.3	90.7 / 77.8	87.9 / 8.6
no. of generations	18	82	172	855	1944
no. of individuals	15 048	66 481	139 476	689 510	813 204
testing time [s]	62	570	1225	7489	12339
branch cover	ET / RT	ET / RT	ET / RT	ET / RT	ET / RT
coverage [%]	100 / 50	100 / 61.8	100 / 11.9	83.7 / 73.5	72.2 / 7.6
no. of generations	16	74	206	1610	3224
no. of individuals	12 944	60 046	166 298	1 298 394	2 600 249

tests are few. Only an interface specification of the system under test is needed to guarantee the generation of valid input values. For structural testing the source code of the test object is also required. The evolutionary test is universally applicable because it adapts itself to the system under test.

In order to guarantee an efficient overall test, the test control of the evolutionary test environment evaluates every individual with regard to every partial aim that has not been reached. Partial aims reached purely by chance are thus identified immediately. Individuals suitable for one or more partial aims are noted, stored, and used as seeds at the optimization of these partial aims. The processing sequence of the partial aims is guided by the quality of the available initial values. In this way, the test quickly achieves the highest possible coverage. Experiments utilizing this strategy have proved successful, and the overall testing procedure has been accelerated considerably.

Evolutionary testing has already produced very good results in the application field. Therefore, evolutionary testing seems to have the potential to increase the effectiveness and efficiency of existing test processes. Evolutionary tests thus contribute to quality improvement and to reduction of development costs.

The users applying evolutionary testing in industrial practice can not be assumed to have any knowledge of evolutionary algorithms. Thus, the selection of evolutionary algorithms employed for the testing of a system needs to take place without the participation of the users. By means of extended evolutionary algorithms (Section 5.2), which combine global and local search procedures in several subpopulations, robust optimization results are obtained for a large number of different testing tasks.

Since research in the field of evolutionary computation is carried out intensively world-wide, further improvements of the search techniques can be expected in the future. Evolutionary testing could directly benefit from such improvements by incorporating new search techniques into test data generation, thus leading to a further increase in the effectiveness and efficiency of the tests.

At present, statement tests, branch tests, condition tests, and segment tests can be applied. Work on multiple-condition testing is drawing to a close. The test environment will also be extended for structural testing of object-oriented Java programs. Furthermore, a visualization component for observing the testing progress will be included and the distribution of tests to several computers will be supported.

References

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

[2] *Buhr, K.*: Einsatz von Komplexitätsmaßen zur Beurteilung Evolutionärer Testbarkeit (Complexity Measures for the Assessment of Evolutionary Testability). Diploma Thesis, Technical University Clausthal, 2001.

[3] Capability Maturity Model for Software, Software Engineering Institute, Carnegie Mellon University.

[4] *Korel, B.*: Automated Test Data Generation. IEEE Transactions on Software Engineering, vol. 16 no. 8 pp.870-879, 1990.

[5] IEC 65A Software for Computers in the Application of Industrial Safety-Related Systems (Sec 122).

[6] *Jones, B. F., Eyres, D. E. and Sthamer, H.-H.*: A Strategy for using Genetic Algorithms to Automate Branch and Fault-based Testing. The Computer Journal, vol. 41, no. 2, pp. 98 – 107, 1998.

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

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

[9] *McGraw, G., Michael, C., Schatz, M.*: Generating Software Test Data by Evolution. Technical Report RSTR-018-97-01, RST Corporation, Sterling, Virginia, USA, 1998.

[10] *Pohlheim, H.*: GEATbx - Genetic and Evolutionary Algorithm Toolbox for Matlab. http://www.geatbx.com/, 1994-2002.

[11] *Pohlheim, H.*: Evolutionäre Algorithmen - Verfahren, Operatoren, Hinweise aus der Praxis. Berlin, Heidelberg: Springer-Verlag, 1999. http://www.pohlheim.com/eavoh/

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

[13] RTCA/DO-178B Software Considerations in Airborne Systems and Equipment Certification.

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

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

[16] *Watkins, A.*: A Tool for the Automatic Generation of Test Data Using Genetic Algorithms. Proceedings of the Software Quality Conference '95, Dundee, Great Britain, pp. 300-309, 1995.

[17] *Wegener, J.*: Evolutionärer Test des Zeitverhaltens von Realzeit-Sytemen (Evolutionary Testing of the Temporal Behaviour of Real-Time Systems). Shaker Verlag, 2001.

[18] *Wegener, J.; Baresel, A.; Sthamer, H.*: Evolutionary Test Environment for Automatic Structural Testing. Information and Software Technology, Special Issue devoted to the Application of Metaheuristic Algorithms to Problems in Software Engineering, vol. 43, pp. 841 – 854, 2001.

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