Evaluating Evolutionary Testability with Software-Measurements

Frank Lammermann, André Baresel, and Joachim Wegener

DaimlerChrysler AG, Alt-Moabit 96a, 10559 Berlin, Germany
{Frank.Lammermann, Andre.Baresel,
Joachim.Wegener}@DaimlerChrysler.com

Abstract. Test case design is the most important test activity with respect to test quality. For this reason, a large number of testing methods have been developed to assist the tester with the definition of appropriate, error-sensitive test data. Besides black-box tests, white-box tests are the most prevalent. In both cases, complete automation of test case design is difficult. Automation of black-box test is only meaningfully possible if a formal specification exists, and, due to the limits of symbolic execution, tools supporting white-box tests are limited to program code instrumentation and coverage measurement. Evolutionary testing is a promising approach for automating structure-oriented test case design completely. In many experiments, high coverage degrees were reached using evolutionary testing. In this paper we shall investigate the suitability of structure-based complexity measures to assess whether or not evolutionary testing is appropriate for the structure-oriented test of given test objects.

1 Introduction

A large number of today's products are based on the deployment of embedded systems. Examples can be found in nearly all industrial areas, such as 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 software development for embedded systems, analytical quality assurance techniques must also be employed intensively besides constructive methods for the specification, design and implementation of the systems. In practice, the most important analytical quality assurance measure is dynamic testing. Testing is the only procedure which allows for the examination of dynamical system behavior in the real application environment. Test case design is the most important test activity, since the type, scope, and quality of the test are determined by selecting feasible test cases. Evolutionary testing is a promising approach for the automation of structure oriented test case design [1, 2, 3, 4, 5]. As evolutionary tests are based on the use of heuristic search methods, it is difficult to assess whether the use of evolutionary tests for a concrete test object is promising or not – that is, whether or not the evolutionary test

K. Deb et al. (Eds.): GECCO 2004, LNCS 3103, pp. 1350–1362, 2004.
© Springer-Verlag Berlin Heidelberg 2004

is able to determine test cases which achieve a high code coverage. We term a test object's suitability for evolutionary testing its *evolutionary testability*.

In this paper we investigate the suitability of structure-based software measures for predicting the evolutionary testability of a test object. If it is possible to establish a relationship between software measures and evolutionary testability, then the evolutionary test could be tailored to the needs of the individual test objects in order to achieve the highest degree of efficiency possible.

This paper is arranged as follows: In the second section, the basics for the automation of test case design for structure-oriented test procedures with evolutionary tests are provided. In the third section, the software measures investigated are briefly presented. Section 4 provides an overview of the results. The causes of the results and the weaknesses of the analyzed software measures are examined in more detail. Section 5 provides an evaluation of these results. The paper closes with a short summary in the sixth section.

2 Test Case Design

In the context of test case design, those test cases are defined with which the testing of the system is to be executed. Existing test case design methods can essentially be differentiated into black-box tests and white-box tests. In the case of black-box tests, test cases are determined from the specification of the program under test, whereas, in the case of white-box tests, they are derived from the internal structure. In both cases, complete automation of the test case design is difficult. Automation of the black-box test is only meaningfully possible if a formal specification exists, and, due to the limits of symbolic execution, tools supporting structure-oriented tests are limited to program code instrumentation and coverage measurement. The aim of applying evolutionary testing to structure-oriented test case design is the generation of a quantity of test data, leading to the best possible coverage of the structural test criterion under consideration. In the case of this test case generation method, it is only possible to determine the input values. Corresponding expected values must be defined by the tester him or herself.

Evolutionary testing is characterized by the use of meta-heuristic search techniques for test case generation. 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 fulfil the respective test aim. Due to the non-linearity of software (if-statements, loops etc.) the conversion of test problems into optimization tasks mostly results in complex, discontinuous, and non-linear search spaces. 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. [1, 3, 5, 6, 7].

2.1 Application of Evolutionary Testing to White-Box Testing

In order to automate test case design for white-box testing with the aid of evolutionary algorithms, the test is divided into test aims. Each test aim represents a program structure that requires execution to achieve full coverage, e.g. for simple condition testing each program condition represents two test aims: evaluating the condition as True and as False. The fitness function is minimized during optimization. If an individual obtains a fitness value of 0, a test datum is found which fulfils all branching conditions in the way required to reach the current test aim. The evolutionary test proceeds to the next test aim. For a more detailed description of the fitness functions refer to [5]. Further definitions of fitness functions for evolutionary structure tests are contained in [1, 3].

3 Structure-Based Software Measures

In the case of the structure-oriented tests considered here, a program has a high level of *evolutionary testability* if and only if two requirements are met during the evolutionary test for the chosen test criterion. In the first instance, a high level of coverage must be achieved and in the second, a low number of test data necessary. These are exactly the requirements in which we are interested for the possible application of evolutionary tests. If it were possible to reliably predict the evolutionary testability of a program or that of its individual test aims, one would be able to decide, before the test, which test termination criteria should be chosen for the individual test aims and whether or not an evolutionary structure test would be of any use for the program. Furthermore, the scope of the search and the selection of the evolutionary operators used in dependence on the qualities of the test object could be changed. Therefore, we shall look at the complexity measures *Number of Test Aims, Executable Lines of Code*, *Halstead's Vocabulary*, *Halstead's Length* [8], C*yclomatic Complexity* [9], *Myers Interval* [10], and *Nesting Level Complexity* [11], all of which allow statements as to the structural properties of programs to be made.

Executable Lines of Code. When determining the Executable Lines of Code (ELOC) the data-flow and control-flow properties of the software examined are not taken into consideration. It only contains lines that contain executable statements.

Halstead's Length and Vocabulary. The software measure Halstead's Length (HALL) is based on counting the number of operands and operators as well as their number of uses in the examined software. When calculating Halstead's Vocabulary (HALV), in contrast, the number of different operators and operands is also taken into account.

Cyclomatic Complexity. Cyclomatic Complexity (CYC) is defined as the number of edges of the programs control-flow graph minus the number of its nodes plus two times the number of its linked components.

Myers Interval. Myers Interval (MI) is an extension of Cyclomatic Complexity, which takes the complexity of the branching conditions more accurate into account. The value of this metric is the sum of the number of logical operators AND and OR in the conditional expressions of the program investigated.

Nesting Level Complexity. Nesting Level Complexity (NLC) assesses the complexity of a software with regard to the nesting level of its conditions and statements.

Number of Test Aims. The aim of the evolutionary structure test is to find a set of test data with which every test aim of the test object is reached at least once. The software measure Number of Test Aims (NTA) can thus be used as a means of estimating the test effort, which would be expected to increase with the number of test aims.

4 Experiments

Source text and structure-based software measures do not seem to be able to sufficiently express the evolutionary testability of a test object. In [12] Buhr showed, using 40 test objects, that the seven above-mentioned software measures were not adequately suited to this purpose. Buhr was unable to establish a sufficient connection between these software measures and the level of coverage achieved, to justify the use of software measures to reach conclusions regarding evolutionary test behavior. The cause is stated as being the insignificant connection between the structural properties evaluated by the software measures and the properties which it is necessary to describe in order to judge evolutionary testability. To investigate the difficulties which conventional software measures have in evaluating evolutionary testability we would like to take a closer look at the experiments.

4.1 Test Preparation

A large number of different test objects was chosen, which spanned a broad spectrum of different program complexities and originated from different application areas: mathematical calculations (both control-related tasks and the execution of string and character operations) and components from automotive and motor electronics. They possessed an appropriate value spectrum for each software measure selected: ELOC varied from 7 to 320, HALV reached values between 25 and 949, HALL between 93 and 9138. CYC in one area differed from between 3 to 50, MI reached 0 to 47, NLC varied from 1 to 17 and NTA from 4 to 132.

The minimal multiple-condition coverage test was chosen as a structure test criterion in the experiments because in this test the coverage level reflects the percentage of non-accomplishable test cases exactly. The branch coverage test can lead to the problem that the number of test goals not achieved does not comply with the number of non-executable commands and thus some conditions may be optimized several times. As the parameter settings of the evolutionary algorithms influence the coverage

level achieved to a large extent, the evolutionary parameters were kept constant during all the experiments. This ensures that the results can be compared. For all the experiments, the population size was set at 300 individuals: 6 subpopulations, each made up of 50 individuals. Each subpopulation deployed different evolutionary algorithms, which, in turn, pursued different search strategies and competed with each other. Fitness assignment took place proportionally, selection was carried out by means of Stochastic Universal Sampling [13] and the type of recombination used was Mühlenbein und Schlierkamp-Voosen's discrete recombination [14]. On average, for each individual, a variable was mutated and executed with the aid of the mutation of real variables. During reinsertion the generation gap was 90%, i.e. the next generation consisted of 10% parent individuals and 90% offspring. Five test runs per test object were carried out for a total of 13 selected test objects using these settings.

When choosing coverage level and the generations, only those test goals from the control flow graph were taken into account which were executable and which the evolutionary structure test, due to the definition of its fitness function, could reach with a sufficiently long execution time.

Table 1. Minimal multiple-condition coverage test of the 13 test objects investigated. The number of evolutionarily possible test goals, the average coverage level reached during this process and the average number of generations required are provided as results

Test object No.	Test object name	evol. possible test goals	Coverage reached in %	No. of average generations
1	blockerkennung	27	85,9	533
2	bnldev	24	98,3	248
3	einklemmschutz	10	90,0	308
4	firstJan	4	100,0	28
5	function_hhs	56	100,0	322
6	gcd1	5	100,0	79
7	gcd2	12	100,0	18
8	hail	16	100,0	70
9	kindersicherung	52	88,9	3.175
10	leap	4	100,0	47
11	mzuef	126	100,0	23
12	powi	22	100,0	35
13	tuermodul	97	80,0	8.612

In order to be able to evaluate the quality of software measures with regard to their ability to judge evolutionary testability, one can compare them with real measurements of evolutionary testability (table 1). It is possible to do this by sorting test objects according to their evolutionary testability measured. The further in front a test object is in the sorting, the higher its evolutionary testability will be. For this reason, weighting is carried out firstly according to the coverage reached and then according to the number of generations required (see sect. 3). This kind of sorting will be called *evolutionary sorting* in the following. Thus it becomes possible to evaluate various other measures with regard to their suitability to measuring evolutionary testability, depending on how far their sorting differs from the evolutionary sorting.

4.2 Average Divergence in the Case of Sortings

So as to be able to assess the prediction quality of a software measure regarding evolutionary testability we need to have a look at the divergence of the test objects sorted according to the measure from test objects sorted in an evolutionary way. The divergence describes the sum of distances of all positions of the same test objects in two different sortings. The less the test objects which are sorted on the basis of a certain software measure diverge from the evolutionary sorting method, the better the influence on the quality of the predicted evolutionary testability will be.

The *average divergence* of a random sorting from an evolutionary sorting or any other sorting could help us in evaluating the quality of the software measures. If a sorting of a specific software measure diverges from the evolutionary sorting in such a way that it is close to the average divergence to be expected, this means that it does not contain any suitable prediction quality. In order to determine the average divergence $\mathrm{div}_{\varnothing}(n)$ we will now have a closer look at two sortings of n elements:

Let $S_n = \{x_1, x_2, \ldots, x_n\}$ and $S_n' = \{x_1', x_2', \ldots, x_n'\}$ be any two sortings of an ordered set A, whose n elements are clearly distinguishable with regard to one variable relation. Now, the average overall divergence of S_n from S_n' is calculated from the sum of the average divergences of all the elements.

$$\mathrm{div}_{\varnothing}(x_1) = \frac{\sum_{i=0}^{n-1} i}{n} \tag{1}$$

applies to the first element x_1 of the sorting S_n since each of the positions receives the same probability $\frac{1}{n}$. The same goes for the last element x_n, i.e. $\mathrm{div}_{\varnothing}(x_1) = \mathrm{div}_{\varnothing}(x_n)$.
To the second and the next to last element of the sorting S_n the following line applies:

$$\mathrm{div}_{\varnothing}(x_2) = \mathrm{div}_{\varnothing}(x_{n-1}) = \frac{1+\sum_{i=0}^{n-2} i}{n} = \frac{\sum_{j=0}^{1} j+\sum_{i=0}^{n-2} i}{n}, \tag{2}$$

and for the third and third to last element:

$$\mathrm{div}_{\varnothing}(x_3) = \mathrm{div}_{\varnothing}(x_{n-2}) = \frac{1+2+\sum_{i=0}^{n-3} i}{n} = \frac{\sum_{j=0}^{2} j+\sum_{i=0}^{n-3} i}{n}. \tag{3}$$

This continues up to the middle elements, to which the following lines apply:

$$\mathrm{div}_{\varnothing}(x_{\frac{n}{2}}) = \mathrm{div}_{\varnothing}(x_{\frac{n}{2}+1}) = \frac{\sum_{j=0}^{\frac{n}{2}-1} j+\sum_{i=0}^{\frac{n}{2}} i}{n} \quad \text{where n is even.} \tag{4}$$

$$\mathrm{div}_{\varnothing}(x_{\lceil \frac{n}{2} \rceil}) = \frac{\sum_{j=0}^{\lfloor \frac{n}{2} \rfloor} j+\sum_{i=0}^{\lfloor \frac{n}{2} \rfloor} i}{n} \quad \text{where n is odd.} \tag{5}$$

After adding up all the average divergences of the individual elements $x_1, x_2, \ldots, x_n$, we get for n even the following overall divergence

$$\mathrm{div}_{\varnothing}(n) = \sum_{i=1}^{n} \mathrm{dev}_{\varnothing}(x_i) = 2 \cdot \left(\frac{\sum_{i=0}^{n-1} i}{n} + \frac{\sum_{j=0}^{1} j + \sum_{i=0}^{n-2} i}{n} + \frac{\sum_{j=0}^{2} j + \sum_{i=0}^{n-3} i}{n} + \cdots + \frac{\sum_{j=0}^{\frac{n}{2}-1} j + \sum_{i=0}^{\frac{n}{2}} i}{n} \right) =$$

$$2 \cdot \frac{\sum_{j=0}^{\frac{n}{2}-1} \sum_{i=0}^{j} i + \sum_{j=\frac{n}{2}}^{n-1} \sum_{i=0}^{j} i}{n} = \frac{2 \cdot \sum_{j=0}^{n-1} \sum_{i=0}^{j} i}{n} \tag{6}$$

and for n odd

$$\mathrm{div}_{\varnothing}(n) = \sum_{i=1}^{n} \mathrm{div}_{\varnothing}(x_i) = 2 \cdot \left(\frac{\sum_{i=0}^{n-1} i}{n} + \frac{\sum_{j=0}^{1} j + \sum_{i=0}^{n-2} i}{n} + \frac{\sum_{j=0}^{2} j + \sum_{i=0}^{n-3} i}{n} + \cdots + \frac{\sum_{j=0}^{\lfloor \frac{n}{2} \rfloor - 1} j + \sum_{i=0}^{\lceil \frac{n}{2} \rceil} i}{n} \right) +$$

$$\frac{\sum_{j=0}^{\lfloor \frac{n}{2} \rfloor} j + \sum_{i=0}^{\lfloor \frac{n}{2} \rfloor} i}{n} = 2 \cdot \left(\frac{\sum_{j=0}^{\lfloor \frac{n}{2} \rfloor - 1} \sum_{i=0}^{j} i + \sum_{j=\lfloor \frac{n}{2} \rfloor}^{n-1} \sum_{i=0}^{j} i}{n} \right) + \frac{\sum_{j=0}^{\lfloor \frac{n}{2} \rfloor} j + \sum_{i=0}^{\lfloor \frac{n}{2} \rfloor} i}{n} = \frac{2 \cdot \sum_{j=0}^{n-1} \sum_{i=0}^{j} i}{n} . \tag{7}$$

Consequently, the following applies in general to the average divergence of an n-element sorting:

$$\mathrm{div}_{\varnothing}(n) = \frac{2 \cdot \sum_{j=0}^{n-1} \sum_{i=0}^{j} i}{n} . \tag{8}$$

From this, it follows that for the 13 remaining test objects $\mathrm{div}_{\varnothing}(13) = 56$. This means that 56 is the value which we get as an average for the divergence from the evolutionary sort sequence, if we arbitrarily sort the test objects without any particular order.

4.3 First Results

If we apply the seven software measures mentioned above to the 13 test objects they are not able to provide us with reliable predictions concerning evolutionary testability – as already noticed in [12]. In table 2 and figure 1 the evolutionary sorting is depicted, while in table 3 and figure 2 the most successful sorting according to NLC is shown. Also, for the evaluation of evolutionary testability the average degree of coverage of the minimal multiple condition coverage and the number of generations needed on average to achieve the evolutionary possible test aims was stated for every test object. The best possible sorting on the basis of the software measure NLC possesses a test object (no. 11), whose position within a sorting of only 13 test objects nonetheless diverges from the evolutionary sorting by six positions. Furthermore, there are two test objects (no. 5 and no. 6), which each diverge by five positions.

Table 2. Evolutionary sorting of the test objects

Test object no.	Test object name	Coverage in %	Number of generations
7	gcd2	100,0	18
11	mzuef	100,0	23
4	firstJan	100,0	28
12	powi	100,0	35
10	leap	100,0	47
8	hail	100,0	70
6	gcd1	100,0	79
5	function_hhs	100,0	322
2	bnldev	98,3	248
3	einklemmschutz	90,0	308
9	kindersicherung	88,9	3.175
1	blockerkennung	85,9	533
13	tuermodul	80,0	8.612

Table 3. Sorting of test objects on the basis of NLC

Test object no.	Test object name	Coverage in %	Number of generations	NLC
7	gcd2	100,0	18	23
6	gcd1	100,0	79	23
4	firstJan	100,0	28	46
10	leap	100,0	47	46
12	powi	100,0	35	70
8	hail	100,0	70	70
3	einklemmschutz	90,0	308	70
11	mzuef	100,0	23	93
2	bnldev	98,3	248	93
9	kindersicherung	88,9	3.175	93
1	blockerkennung	85,9	533	139
13	tuermodul	80,0	8.612	139
5	function_hhs	100,0	322	395

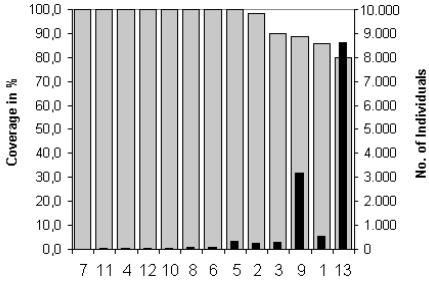

Fig. 1. Evolutionary sorting of the test objects

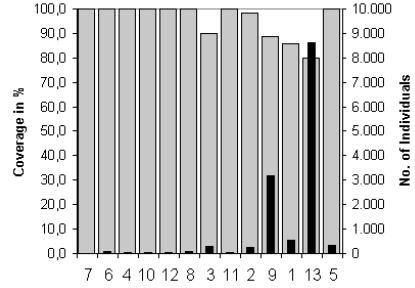

Fig. 2. Sorting of test objects on the basis of NLC

Table 4 shows us a closer quantification of the individual sortings sorted on the basis of software measures. Here we can find the divergences of the test objects which were sorted according to the seven chosen software measures from the evolutionary sorting sequence. It is clear that the best possible sorting of the test objects, based on the complexity measure NLC and provides us with a value of 24, is not very successful when compared with an average divergence value of 56. Additionally, the worst possible sortings which were sorted according to the software measures HALF and HALL show a value of 50 which is almost the result of random sorting.

Table 4. Divergences of software measures compared to evolutionary sorting

Software measure	ELOC	HALF	HALL	CYC	MI	NLC	NTA
Divergence	46	50	50	44	32	24	48

4.4 Software Measure Combinations

A further approach consists of combining different software measures in order to obtain reliable statements as to their evolutionary testability. It is possible that the weaknesses of individual software measures may be compensated for by the combination of several software measures.

The software measures are normalized, thus allowing us to compare their different values. It then has to be investigated which combination and weighting of the individual software measures represents the best possible measure for describing evolutionary testability. During this investigation the seven software measures together with the divergence resulting from the evolutionary sorting span an eight-dimensional search space, whose minimum has to be optimized.

The evolutionary algorithm lends itself to this because it is particularly well suited to optimizing a multi-dimensional search space using a corresponding fitness function [15]. Fitness assignment occurs in proportion to the divergence of the sorting sequence, resulting from the weightings of the software measure combinations, to the evolutionary sorting. In this way, the evolutionary algorithm will favor those software measure combinations which best describe evolutionary testability.

Table 5. Best divergences and weightings of the software measure combinations

Test no.	Divergence	ELOC	HALV	HALL	CYC	MI	NLC	NTA
1	30	0.0010	0.0033	0.0480	0.0011	0.9944	0.4167	0.0730
2	30	0.0505	0.0001	0.0123	0.0000	0.9092	0.3624	0.0295
3	30	0.0017	0.0768	0.0036	0.0075	0.8927	0.4312	0.0094
4	30	0.0000	0.0371	0.0000	0.0305	0.9372	0.4646	0.0350
5	30	0.0702	0.0121	0.0206	0.0012	0.9881	0.4167	0.0156
6	30	0.0449	0.0004	0.0274	0.0000	0.9092	0.3624	0.0295
7	30	0.0020	0.0507	0.0036	0.0108	0.9690	0.4164	0.0453
8	30	0.0000	0.0191	0.0082	0.0000	0.9632	0.4646	0.0350
9	30	0.0064	0.0120	0.0416	0.0011	0.9872	0.5204	0.0307
10	30	0.0016	0.0000	0.0036	0.0075	0.8873	0.4157	0.0355

During the optimization, values in the interval [0, 1] were used for the weighting of the individual software measures. The sum of the respective software measure weightings produced a new measure, from which a new sorting sequence also resulted. It turned out that, with the exception of MI and NLC all the values of the software measures always lay between 0 and 0.08 (table 5). In contrast, the values for MI ranged between 0.88 and 1 and those for NLC between 0.36 and 0.53. This leads us to surmise that NLC and MI primarily contribute to optimal sorting. However, if one considered these two software measures exclusively for an optimization run, it would not be possible to attain below 30 unless the proportion of MI became vanishingly low in comparison to NLC (< approx. 0,2%), which alone managed to achieve a divergence of 24. In no run was it possible to achieve a lower value than this.

If we compare table 4 and table 5, the result is astonishing: The best value of the software measure combinations obtains a divergence of 30 and is presented in table 5 with the respective weightings of the software measures. In not one of the ten test

runs was any of the combinations able to achieve a divergence as low as the best software measure NLC alone, which obtained a divergence of 24. Combinations of software measures obviously do not lead to improvements compared to individual software measures. It seems that combinations, however they are weighted, weaken the predictions of individual software measures with regard to evolutionary testability.

4.5 Eliminating Disturbing Influences

Other influences resulting from the test objects investigated also play a decisive role in determining evolutionary testability. Buhr [12] believes that it is not the structure but rather the data flow properties of a test object which primarily influence its evolutionary testability. This assumption is based on the localization of different disturbing influences – disturbing influences which could occur during the execution of the evolutionary structure test. Examples are incrementors, i.e. local static variables which act as timers and only make it possible to achieve a condition by repeating program calls very often; floating point optimizations whose randomly created optimization steps might possibly be too large to reach a certain constant value correctly; flags [16] or side-effects [17]. These so strongly disturb the evolutionary testability that characteristics described by source code or structure-based software measures no longer have any sufficient effect.

If we eliminate those test objects which have disturbing influences on the evolutionary structure test concerning test goal reachability, this would leave us with only seven of the 13 test objects described. With these seven test objects some of the software measures analyzed provide quite reliable predictions about evolutionary testability. As a measure for the quality of a reliable prediction the divergence from the evolutionary sorted test objects was again chosen. In table 6 the divergences of all software measures for this test series are depicted, in table 7 and figure 3 the evolutionary sorting is shown, and in table 8 and figure 4 one can find the most successful sorting according to CYC.

With seven test objects the average divergence $div_{\varnothing}$ achieves a value of 16. Nevertheless, the sorting based on CYC surpasses this by a factor of 4. Even if we combine the software measures with the help of the evolutionary algorithm, the result of 6 reached by test objects without disturbing influences on the evolutionary structure test, is not better than the most successful sorting of individual software measures.

Table 6. Divergences of software measures compared to evolutionary sorting of test objects without problem cases

Software measure	ELOC	HALF	HALL	CYC	MI	NLC	NTA
Divergence	6	10	10	4	6	6	6

Table 7. Evolutionary sorting of test objects without problem cases

Test object no.	Test object name	Coverage in %	Number of generations
4	firstJan	100,0	28
10	leap	100,0	47
8	hail	100,0	70
6	gcd1	100,0	79
3	einklemmschutz	90,0	308
9	kindersicherung	88,9	3.175

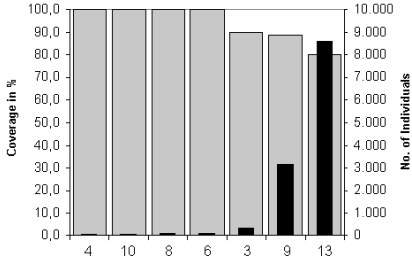

Fig. 3. Evolutionary sorting of test objects without problem cases

Table 8. Sorting of test objects without problem cases based on CYC

Test object no.	Test object name	Coverage in %	Number of generations	CYC
4	firstJan	100,0	28	22
10	leap	100,0	47	22
6	gcd1	100,0	79	22
3	einklemmschutz	90,0	308	44
8	hail	100,0	70	59
9	kindersicherung	88,9	3.175	169

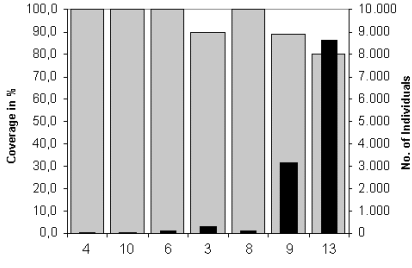

Fig. 4. Sorting of test objects without problem cases based on CYC

5 Evaluation

On the basis of the experiments it could be shown that by using source code or structure-based software measures it was only possible to make mediocre predictions as to evolutionary testability, which complies with Buhr's findings. Characteristics of the test objects exist, which are not sufficiently depicted by these software measures, even if the test objects are free of disturbing influences regarding the reachability of the test aim. It is questionable how strongly the differing weightings of the test objects within the sortings would affect the results. To make better statements, extensive test series which use an application area which is as broad as possible need to be executed.

Moreover, slight source code modifications such as, for example, greater limitations in the test objects' conditions hardly influence the software measures or do not do so at all. They can, however, have a powerful influence on evolutionary testability. This is illustrated by the following example:

```
long w, x, y, z;
if (w == 0 && x == 0 && y == 0 && z == 0) { // testaim
```

In the experiment, this artificially created test object obtained an average coverage of 40% for five test runs. This low coverage is a consequence of the subconditions' substantial limitations. The evolutionary structure test is already overstrained starting from two subconditions to be fulfilled together, producing 200 test data generations. The ratio of positive to negative test data is 1 to $1.84 * 10^{19}$ for two subconditions and around 1 to $3.40 * 10^{38}$ for all four conditions. Only for one or no subconditions to be fulfilled, could a successful test datum be found. The test object thus possesses low evolutionary testability, whereas the low software measures would lead one to expect high evolutionary testability (except for MI and NTA the evaluation of the software measures is always the lowest in this example). In the case of this test object, the source code and structure-based software measures would fail to evaluate evolutionary testability.

6 Conclusion

In order to increase the quality of tests and to reduce the development costs for software-based systems, test methods are called for which support a complete test and which are, to a large extent, automatable. The evolutionary test lends itself to this, because it supports fully automated test case generation during structure tests.

So as to estimate the evolutionary testability of test objects we checked for a possible connection between different software measures and the execution of the evolutionary test. We looked at the following source code and structure-based software measures: *Number of Test Aims, Executable Lines of Code*, *Halstead's Vocabulary*, *Halstead's Length*, C*yclomatic Complexity*, *Myers Interval*, *Nesting Level Complexity*, and *Number of Test Goals*. The values measured were compared to the evolutionary testability of test objects of varying complexity. After the elimination of disturbing influences within test objects which would hinder the execution of the evolutionary test, the structure-based software measure C*yclomatic Complexity* was able to generate the best forecasts for the evolutionary testability to be expected.. *The sorting of the test objects based on this software measure came quite close to the sorting based on evolutionary testability and surpassed an average sorting by a factor of 4.*

It was possible to identify cases in which the software measure C*yclomatic Complexity would fail, i.e. there are test objects whose evolutionary testability is rated incorrectly by the* C*yclomatic Complexity measure (or any other of the source code or structure-based software measures analyzed). The reason for this can be found in the dependency of evolutionary testability on certain structure characteristics of the software to be tested, which none of the software measures analyzed (nor any other known software measure) is capable of expressing.* Thus, future work should concentrate on the development of a software measure which is able to mirror evolutionary testability as exactly as possible.

Acknowledgements. The work described has been performed within the SysTest project. The SysTest project is funded by the European Community under the 5th Framework Programme (GROWTH), project reference G1RD-CT-2002-00683.

References

1. Sthamer, H.: The Automatic Generation of Software Test Data Using Genetic Algorithms, PhD Thesis, University of Glamorgan, Pontyprid, Wales, Great Britain (1996)
2. Tracey, N., Clark, J., Mander, K., McDermid, J.: An Automated Framework for Structural Test-Data Generation, Proceedings of the 13th IEEE Conference on Automated Software Engineering, Hawaii, USA (1998)
3. Pargas, R., Harrold, M., Peck, R.: Test-Data Generation Using Genetic Algorithms, Software Testing, Verification & Reliability, vol. 9, no. 4 (1999) 263–282
4. Michael, C., McGraw, G., Schatz, M.: Generating Software Test Data by Evolution, IEEE Transactions on Software Engineering, vol. 27, no. 12 (2001) 1085–1110
5. Wegener, J., Baresel, A., Sthamer, H.: Evolutionary Test Environment for Automatic Structural Testing, Information and Software Technology, vol. 43 (2001) 841–854
6. Wegener, J., Grochtmann M.: Verifying Timing Constraints of Real-Time Systems by means of Evolutionary Testing, Real-Time Systems, vol. 15, no. 3 (1998) 275–298
7. Jones, B., Eyres, D., Sthamer, H.: A Strategy for using Genetic Algorithms to Automate Branch and Fault-based Testing, The Computer Journal, vol. 41, no. 2 (1998) 98–107
8. Halstead, M.: Elements of Software Science, Prentice Hall, New York, USA (1977)
9. McCabe, T.: A Complexity Measure, IEEE Transactions on Software Engineering, vol. 2, no. 12 (1976) 208–220
10. Myers, G.: An Extension to the Cyclomatic Measure, ACM SIGPLAN Notices, vol. 12, no. 10, 61–64
11. QA C Source Code Analyser – Command Line, Programming Research Ltd. (1974)
12. Buhr, K.: Einsatz von Komplexitätsmaßen zur Beurteilung Evolutionärer Testbarkeit (Complexity Measures for the Assessment of Evolutionary Testability). Diploma Thesis, Technical University Clausthal (2001)
13. Baker, J.: Reducing Bias and Inefficiency in the Selection Algorithm. Proceedings of the 2nd International Conference on Genetic Algorithms (ICGA '87), Cambridge, Massachusetts, USA (1987) 14–21
14. Mühlenbein, H., Schlierkamp-Voosen, D.: Predictive Models for the Breeder Genetic Algorithm: I. Continuous Parameter Optimization, Evolutionary Computation, vol. 1, no. 1 (1993) 25–49
15. Pohlheim, H.: Evolutionäre Algorithmen. Verfahren, Operatoren und Hinweise für die Praxis, Springer-Verlag, Berlin (2000)
16. Harman, M., Hu, L., Hierons, R., Baresel, A., Sthamer, H.: Improving Evolutionary Testing by Flag Removal, Proceedings of the Genetic and Evolutionary Computation Conference, New York, USA (2002)
17. Harman, M., Hu, L., Hierons, R., Munro, M., Zhang, X., Dolado, J., Otero, M., Wegener, J.: A Post-Placement Side-Effect Removal Algorithm, Proceedings of the IEEE International Conference on Software Maintenance, Montreal, Canada (2002)