Using Evolutionary Algorithms for the Unit Testing of Object-Oriented Software

Stefan Wappler
DaimlerChrysler AG, Research and Technology
Alt-Moabit 96a, D-10559 Berlin, Germany
Phone: +49 30 39982 358
stefan.wappler@daimlerchrysler.com

Frank Lammermann
DaimlerChrysler AG, Research and Technology
Alt-Moabit 96a, D-10559 Berlin, Germany
Phone: +49 30 39982 272
frank.lammermann@daimlerchrysler.com

ABSTRACT

As the paradigm of object orientation becomes more and more important for modern IT development projects, the demand for an automated test case generation to dynamically test object-oriented software increases. While search-based test case generation strategies, such as evolutionary testing, are well researched for procedural software, relatively little research has been done in the area of evolutionary object-oriented software testing. This paper presents an approach with which to apply evolutionary algorithms for the automatic generation of test cases for the white-box testing of object-oriented software. Test cases for testing object-oriented software include test programs which create and manipulate objects in order to achieve a certain test goal. Strategies for the encoding of test cases to evolvable data structures as well as ideas about how the objective functions could allow for a sophisticated evaluation are proposed. It is expected that the ideas herein can be adapted for other unit testing methods as well. The approach has been implemented by a prototype for empirical validation. In experiments with this prototype, evolutionary testing outperformed random testing. Evolutionary algorithms could be successfully applied for the white-box testing of object-oriented software.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging — *Test coverage of code, Testing tools*

General Terms

Verification

Keywords

object-oriented testing, evolutionary testing, chaining approach, multi-level optimization, automated test case generation

GECCO'05, June 25–29, 2005, Washington, DC, USA.

1. INTRODUCTION

Evolutionary algorithms have been applied successfully for the unit testing of procedural software ([5, 7], referred to as *conventional evolutionary testing*). Hence, it could be expected that they are equally well-suited for the unit testing of object-oriented software (referred to as *object-oriented evolutionary testing*). The scope of conventional evolutionary testing is to find *test data* which serves as input data for the unit under test. In contrast, with object-oriented evolutionary testing, the evolutionary search aims at producing complete *test programs* because input data is by itself not sufficient to execute the test (see [8] and section 2): a test case must also describe how to create the objects participating in the test and how to put them into the proper state for the test goal[1] to be met. An objective function for object-oriented evolutionary testing must evaluate a test program according to its ability to meet a given test goal. Thereby, it must take into account the state behavior of the objects participating in the test. The approaches for the objective functions for conventional evolutionary testing [1, 6] must be adapted to fit the needs of object orientation.

This paper presents an approach for the automatic generation of test programs for object-oriented unit testing using universal evolutionary algorithms. *Universal evolutionary algorithms* are evolutionary algorithms provided by popular toolboxes which are independent from the application domain and offer a variety of predefined, probabilistically well-proven evolutionary operators. The generated test programs can be transformed into test classes according to popular testing frameworks, such as JUnit. In order to employ universal evolutionary algorithms, an encoding is defined to represent object-oriented test programs as basic type value structures[2]. In order to optimize the evolutionary search, multi-level optimizations are considered. The suggested encoding does not prevent the generation of individuals which cannot be decoded into test programs without errors (referred to as *inconvertible individuals*, see 4). Therefore, three measures to be used by the objective function are presented which guide the evolutionary algorithm to generate more and more individuals over time that can successfully be decoded (referred to as *convertible individiuals*).

[1]What a given test goal is depends on the testing method. For instance, in the context of time testing, a test goal is to exceed a certain time limit; in the context of white-box testing, a test goal can be to reach a particular program source code element.
[2]Basic types are built-in types, such as `integer`, `float`, etc.

The investigation carried out by Kim et al. [4] encourages the application of conventional coverage criteria for object-oriented unit testing as well. In this paper, ideas for objective functions for object-oriented white-box testing are presented. A modification of the hybrid approach [6] is suggested. With this modified approach, the evolutionary search for object-oriented test programs can be successful in the presence of flags and state behavior.

This paper is structured as follows: section 2 describes the general structure of object-oriented test programs (the phenotypes). In section 3, the encoding of the phenotypes to form genotypes and the decoding of the genotypes to form phenotypes is discussed. The design of objective functions is described in section 4. The presented approaches can be applied for multiple testing methods. Only section 4.1 consideres one particular testing method, namely white-box testing. Experiments carried out in order to empirically validate the suggested approaches follow in section 5. Before concluding the paper in section 7, related work is dealt with in section 6.

2. PHENOTYPE INDIVIDUALS

A test case for a procedural software unit typically consists of the definition of testing prerequisites, the test data used to execute the unit under test, and the test oracle which decides whether the test is passed or failed. The test data is a set of numerical values (that can be interpreted as character data as well) which are used as either parameter values or read-in data for the function under test. In the case of conventional evolutionary testing, such a set of numerical values is one phenotype individual (see figure 1 a). Within the paradigm

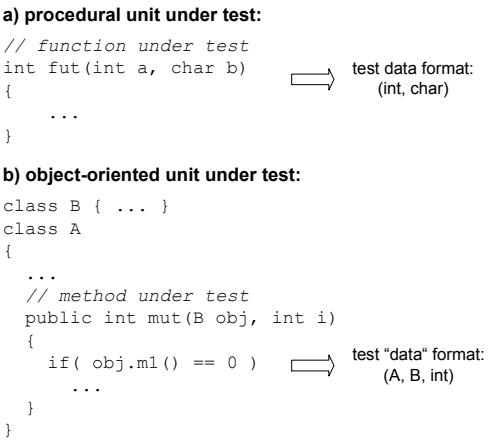

Figure 1: unit test data formats for procedural software (a) and object-oriented software (b)

of object orientation, the major concept is the *object* which possesses attributes (variables) and constructors and methods (procedures). A test case for object-oriented software, does not comprise only numerical test data — a sequence of constructor and method calls is also necessary. This has the following reasons (compare with figure 1 b):

1. Usually, multiple objects are involved in one single test case:

   - At the least, an instance of the class under test is needed.
   - Additional objects which are required (as parameters) for the creation of the object under test and the invocation of the method under test must be available. Again, for the creation of these additional objects, more additional objects can be required. The set of all the classes from which instances can be required is called *test cluster*.

   Therefore, constructor calls must be issued in order to create all the required objects.

2. Depending on the kind of test, the participating objects must be put into special states in order to process the test scenario in the desired way (e. g. when using code coverage criteria, some code elements can only be covered when a certain object is in a particular state). Consequently, method calls must be issued for the test cluster objects.

Thus, a test case for an object-oriented software unit consists of the definition of testing prerequisites, a test program incorporating both test data (as parameter values) and method calls[3], as well as the test oracle. In the background of evolutionary testing, for each test case a test program must be optimized. Consequently, in the context of object-oriented software, phenotype individuals are test programs based on the following production rules[4]:

test_program	::=	{statement;}+
statement	::=	[return_value] {ctor_call|method_call}
return_value	::=	class_name instance_name =
ctor_call	::=	new class_name(parameters)
method_call	::=	{class_name|instance_name}. method_name(parameters)
parameters	::=	[parameter {, parameter}*]
parameter	::=	basic_type_value|instance_name|NULL

Return values are only of interest when they are objects. In such a case, the returned object can serve as a target object or parameter object for succeeding method calls. Basic type return values are irrelevant because basic type parameter values are generated by the evolutionary algorithm. Thus, there is no need for them to be processed further. Also, no control structures are necessary since no branches are needed in a test program and loops are represented by the arbitrary repetition of the same method call(s).

3. GENOTYPE INDIVIDUALS

The evolutionary algorithm must be able to generate test programs as described in the previous section. Typically, universal evolutionary algorithms have no understanding of programs, statements, objects, and so on. Therefore, a means of encoding must be defined which allows the representation of a test program as a basic type value structure (the genotype individual) with which a universal evolutionary algorithm can work.

[3]For simplification, constructors are considered as static methods and are not mentioned explicitely in the following.
[4]Emphasized identifiers designate terminals. The rules for the class names, instance names, and basic type values have been omitted for reasons of simplification. "[]"=option, "{|}"=alternative, "{}+"=repetition with at least one occurrence, "{}*"=arbitrary repetition

In order to define such an encoding, it makes sense to identify structural components of test programs which make a simple numerical representation possible ("divide and conquer"). The collectivity of these single encodings defines the overall encoding of a whole test program. A requirement for the encoding is that it must allow the representation of every conceivable test program.

Each test program can be considered as a sequence of statements $S = (s_1, s_2, ..., s_n)$. A statement consists of the following essential components

- *target object*
- *method*
- *parameters*

It is only this information which needs to be encoded by a genotype individual. Return values are managed implicitely. Since the number of genes for a genotype individual is typically fixed, the user must provide the maximum number of statements.

For the selection of a target object and a method, two genes with integer type are assigned in the genotype individual. When decoding, the integer values of the genes identify the method to be called (value of method-identifying gene G_M) and the target object to be used for the invocation (target-object-identifying gene G_T). Since the methods in a test cluster usually have parameter lists with different lengths, multiple genes G_P must be assigned to represent the parameters for a method. The data type for a G_P depends on the parameter to which it is assigned. If the assignment is not clear when encoding (see 3.3), that data type must be used which would allow for every possible decoding[5]. Consequently, a statement from a test program can be represented by a variable indicating the method to be called (G_M), a variable indicating the target object for which to call this method (G_T), and a number of variables which are used as parameters for the method call (G_P).

Figure 2 shows three possible ways of distributing the necessary genes for a statement among genotype individuals. In alternative 1, all the information for a single test program is encoded in one single genotype individual. As a result, one optimization level is sufficient. A shortcoming of this approach is that the genotype individual usually contains unused genes for the following reason: since it is not known in advance which method will be identified by the method-selecting gene G_M, as many parameter genes G_P must be assigned for each statement as the method with the longest parameter list requires (In figure 2, test programs with three statements are optimized; the longest parameter list has two elements). This shortcoming can be overcome with alternative 2: in the first optimization level, only the methods (call sequence) are optimized. The necessary target-object-identifying genes and parameter genes can be derived from each sequence. These are optimized in the second optimization level (In the figure, it is assumed that G_{M1} identifies

[5]If, for example, the methods and parameters are optimized at the same time, the data type for a parameter at signature position x must allow for the interpretation of the data type's value as a value of any parameter data type at position x for any test cluster method, since it is not clear in advance which method will be "selected" during the evolutionary optimization.

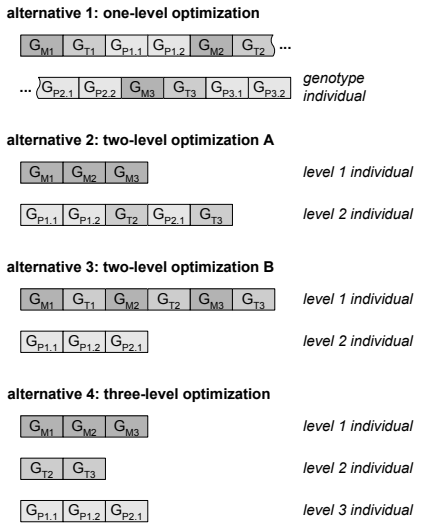

Figure 2: encoding alternatives; sample

a constructor or static method which requires two parameters, G_{M2} identifies a method which requires one parameter, and G_{M3} identifies a method which requires no parameter). Alternative 3 is a modified version of alternative 2: more information (the target-object-identifying genes) is put into the individuals of the first optimization level to increase the expressiveness of the objective values of this level. In alternative 4, each structural component is optimized in a separate optimization level. This allows for an exhaustive search in the overall search space.

Figure 3 shows the workflows for the alternatives. For alternative 1, the flow is similar to that of conventional evolutionary testing (compare [7]). For the other alternatives, another complete optimization is carried out in order to evaluate one single individual. A constant objective value over a predefined number of generations is a promising termination criterion for the "inner" optimizations . If, for instance, after 15 generations no improvement of the objective value can be achieved, the inner optimization terminates. The best objective value achieved by an inner optimization is used as the objective value for an individual of the corresponding outer optimization. In the following sections, the encoding and decoding of the components *methods*, *target objects*, and *parameters* is described in detail. For reasons of simplicity, only one single statement is considered. The encoding and decoding of a test program is a sequence of encodings and decodings of this program's statements. The encoding is performed before the optimization in order to define the format of the genotype indidivuals. The genes involved and their value domains must be defined. The decoding is carried out by the objective function when an individual is evaluated.

3.1 Encoding and decoding of methods

Methods are encoded by serially numbering all the methods in the test cluster classes. In the genotype individual, a gene G_M is assigned whose allele (value) identifies the constructor or method to appear in the test program. The domain D for G_M (labeled I_M) is defined by the maximum

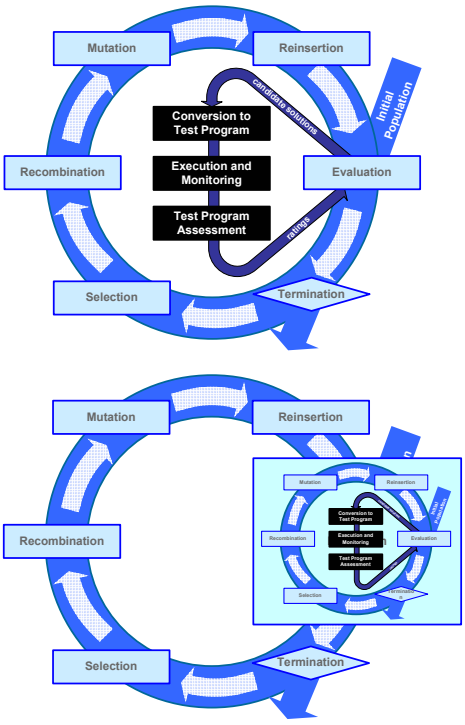

Figure 3: workflow for the alternatives; top picture shows the flow for alternative 1, bottom picture the flow for alternatives 2 and 3, flow for alternative 4 is not illustrated

number of test cluster methods:

$$D(G_M) = I_M = [1, |M_C|] \subset \mathbb{N}$$

whereby $M_C = (m_1, m_2, ..., m_n)$ is the ordered set of methods of the test cluster C.

The decoding of G_M can be described by a function μ which maps each allele of G_M to a method: Let $m_j \in M_C$ be the jth method with $j \in I_M$. Then, the bijective function μ with

$$\mu: \quad I_M \quad \rightarrow M_C \tag{1}$$
$$j \quad \mapsto m_j \tag{2}$$

assigns each method-identifying allele a method by using the allele as a set index.

3.2 Encoding and decoding of target objects

Similar to the encoding of methods, a gene G_T is assigned in the genotype individual to identify the target object for which the method identified by G_M will be called. This is a kind of object reference based on natural numbers. A precise definition of the domain of G_T is only possible if the number of candidate target objects for a statement s_i is known. This number directly depends on the object-creating methods which are called before statement s_i is called. Preceding constructors and methods with an object as a return value exactly define the number of objects which can serve as targets for a particular statement. Hence, for the encoding alternatives 2 to 4, the value domain $D(G_T)$ can be defined for each statement $s_i \in S$ by

$$D(G_{T_i}) = I_{T_i} = [1, |O_{c,i}|] \subset \mathbb{N}$$

whereby $c \in C$ is the required target object class, i is the index of the current statement, and $O_{c,i} = (o_1, o_2, ..., o_n)$ is the ordered set of objects which are instances of class c and have been created by the statements s_1 to s_{i-1}. When using the encoding alternative 1 (see figure 2), it is not clear in advance, how many candidate target objects will be available for statement s_i. Consequently, the domain of G_{T_i} cannot be precisely defined beforehand. A simple strategy for the domain definition in this case is to use a fixed-size large set:

$$D(G_{T_i}) = I_T = [1, MAX_T] \subset \mathbb{N}$$

When decoding, the alleles must be adjusted to the actual number of candidate target objects. This can be done, for example, by multiplying the alleles with the factor $\frac{|O_{c,i}|}{|I_T|}$. Assuming that one statement creates at most one object[6], the value domain $D(G_{T_i})$ can be defined by

$$D(G_{T_i}) = I_T = [1, i] \subset \mathbb{N}$$

For encoding alternatives 2 to 4 (when the method call sequence is known after the first optimization step), no G_T needs to be assigned in the genotype individual for constructors and static methods because no target objects are required for them.

The decoding of G_T can be described by a function τ, which assigns each allele $t \in I_T$ an object $o \in O_{c,i}$:

$$\tau: \quad I_T \quad \rightarrow O_{c,i} \tag{3}$$
$$t \quad \mapsto o_t \tag{4}$$

whereby o_t is the tth object of the set $O_{c,i}$.

3.3 Encoding and decoding of parameters

For each required parameter, a gene G_P is assigned in the genotype individual. The definition of the domain of a G_P depends on the data type of the parameter it represents: for basic data types such as `integer` or `float`, the data type ranges and precision are used as range and precision for $D(G_P)$. For object-type parameters, a similar object reference mechanism is used as for the target objects: When the number of candidate parameter objects is known, the domain of G_P can be defined precisely:

$$D(G_{P_{i,x}}) = I_{P_{i,x}} = [1, |O^*_{c,i}|] \subset \mathbb{N}$$

whereby $x \in \mathbb{N}$ identifies the position in the signature of the method to be called, and $O^*_{c,i}$ is the ordered set of objects which have been created by the statements s_1 to s_{i-1} and which have the type c or any subtype of c to take polymorphism into account. Whenever a parameter object of a class c is required, any instance of class c and its subclasses $\underline{C_c} \subseteq C$ comes into question ($O^*_{c,i} = \bigcup_{c^* \in \{c\} \cup \underline{C_c}} O_{c^*,i}$[7]). Otherwise, if the number of candidate parameter objects is not known when encoding the parameters, a fixed-size index set I_P can again be used:

$$D(G_{P_{i,x}}) = I_P = [1, MAX_P] \subset \mathbb{N}$$

During decoding (when the number of candidate parameter objects is known), the allele must be adapted to the actual number of objects (e. g. by multiplying with $\frac{|O^*_{c,i}|}{|I_P|}$).

[6] In some cases, methods return an array of objects.

[7] In this case, $A \cup B$ with $A = (a_1, a_2, ..., a_m)$ and $B = (b_1, b_2, ..., b_n)$ be defined as set concatenation $(a_1, a_2, ..., a_m, b_1, b_2, ..., b_n)$.

Analoguously to the decoding of G_M and G_T, a function π can be defined which assigns each allele of G_P either a basic type value or an object reference. If there is a basic type value, no mapping is necessary, the allele can directly be used as a parameter value. In case of object-type parameter, a mapping of the index set $I_{P_{i,x}}$ to object references must take place:

$$\pi : \quad I_{P_{i,x}} \quad \rightarrow O^*_{c,i} \tag{5}$$

$$p \quad \mapsto o_p \tag{6}$$

with o_p as the pth object of the ordered set $O^*_{c,i}$.

4. OBJECTIVE FUNCTIONS

The objective function is used to guide the evolutionary search in order to find the optimum. To this end, it must assign each genotype individual an objective value which is used for fitness assignment. In the area of evolutionary testing, the genotype individuals must be converted to test data — or, in the object-oriented context, to test programs — with which the test is then executed. The execution is monitored and the objective value is calculated from monitoring results. Obviously, an objective function for testing object-oriented software must decode the genotype individuals and create executable test programs. With the encoding suggested in section 3, it is possible that no correct test program can be created out of a genotype individual due to the following reasons:

1. No target object is available for a method call: when trying to create a test program statement s_i by decoding a method-identifying gene G_M, it is possible that no appropriate candidate target objects are available ($O_{c,i} = \emptyset$).

2. A required parameter object is not available for a method call: when trying to create the test program statement s_i by decoding a method-identifying gene G_M, it is possible that no appropriate candidate parameter objects are available ($O^*_{c,i} = \emptyset$).

Because the methods, the target objects, and the parameters are optimized independently from one another (but possibly at the same time), it is possible that not all required objects are available for a method call. This happens, for example, if the first method-identifying gene of a test program identifies neither a constructor nor a static method. Consequently, an objective function must evaluate whether a genotype individual can be decoded into a test program on the one hand (issue 1) and whether the optimization problem can be solved on the other hand (issue 2). Issue 1 is independent from the testing method and will be discussed in detail section 4.1. Issue 2 depends on the testing method. An approach to design objective functions for white-box testing is described in section 4.2.

4.1 Evaluation of inconvertible individuals

As already mentioned, not every genotype individual based on the suggested encoding can be used to create a test program. This is not a problem as long as enough individuals exist which can be used for test program creation. However, if no such individual exists in the current population, the evolutionary algorithm must be guided in such a way as to create more and more convertible individuals with the next generations. Otherwise, no optimization takes place. Additionally, having only a few convertible individuals in a generation delimits the diversity of the following generation[8]. An approach to deal with inconvertible individuals is to try to "repair" them, i. e. to change some alleles according to some rules such that the individuals become convertible. These rules would be quite complex, and for reasons of focusing the discussion, this idea is not considered further in this paper. In the case of inconvertible individuals, the objective value must express the *degree* of failure for the decoding. For this evaluation, the following measures are suggested:

- *number of errors*
- *constructor distance*
- *dynamic error evaluation*

These measures are alternatives, only one of them can be applied during optimization. However, other measures are conceivable as well. A value of 0 is assumed to be the optimum of the objective function. The measure *number of errors* assigns each inconvertible genotype individual the total number of missing objects (either target or parameter objects). The measure *constructor distance* assigns each inconvertible genotype individual the number of statements between the erroneous statement (where an object is missing) and the next statement by which an object is created that matches the class of the missing object. This measure is especially qualified for the application of mutation operators which use swapping of variables. The measure *dynamic error evaluation* is similar to *number of errors*. But, in contrast, the errors are weighted according to their repetition: if an object of class c is missing for statement s_i the measure value is increased by a constant a. And if for statement s_{i+j} with $j > 0$ an object of class c is missing again, then the measure value is only increased by a constant b with $b < a$. When evaluating the lack of a parameter object, polymorphism is taken into account by increasing the measure value only by a constant p whereby $p < a$ when an object of a class is missing whose non-existence or the non-existence of a subclass' object had already been considered. An experimental comparison of the measures follows in section 5

An inconvertible individual must be assigned a worse objective value than an bad convertible individual ("bad" here means: unsuitable for solving the optimization problem or leading to a result far away from the optimum). Therefore, the objective values of all the convertible individuals must be adapted to a predefined interval $[0, A]$ (with 0 assumed to be the optimum and having only positive objective values), whereas the objective values of the inconvertible individuals must be shifted out of this interval. This can be accomplished by the following formula. Ω_R is the resulting objective value, Ω_C is the objective value in the case of a convertible individual (for calculation see next section), and Ω_I is the objective value of an inconvertible individual (the measure value of one of the suggested measures):

$$\Omega_R = \begin{cases} A(1+\varepsilon)^{-\Omega_C} & \textit{for convertible individuals} \\ A + \Omega_I & \textit{for inconvertible individuals} \end{cases}$$

with $0 < \varepsilon < 1$.

[8]Convertible individuals receive relatively good objective values and are mainly selected to produce offspring.

4.2 Evaluation of convertible individuals

The calculation of the objective values for the convertible individuals depends on the selected testing method. In principle, for all dynamic testing methods the converted individual — the test program — is executed and the objective value is calculated according to the monitoring results of the execution. In this section, an approach for designing objective functions applicable to the structure-oriented test, especially the statement coverage test, branch coverage test, and condition coverage test, is presented.

The aim of code-coverage-oriented test methods is to yield test cases with which a high number of code elements are executed. The idea behind the distance-oriented conventional evolutionary structural testing approach [1] is to derive a set of single test goals from the given coverage criterion and to search for test data to reach each of the test goals by using an evolutionary algorithm. This strategy can also be applied to the unit testing of object-oriented software. In addition, the combination of the measures *approximation level* and conditional *distance* can be used to assess the capability of a test program to reach a given test goal and hence guide the evolutionary search.

In some cases, since objects represent state machines, it can be necessary to take into account the state behavior of the objects participating in the test. Otherwise, the evolutionary algorithm could not receive any guidance when considering only the approximation level and conditional distance for objective value calculation. One idea to deal with this issue will be described in section 7. The objective functions used in the following case study are based on the distance-oriented approach [1].

5. CASE STUDY

An implementation of the approach using the encoding alternative 2 in figure 2 in Matlab (in order to use the evolutionary toolbox GEATbx [3]) was used to carry out two kinds of studies:

1. At first, some experiments were performed to examine the applicability and feasibility of the approach. To this end, the results of the evolutionary tests were compared with the results of random tests for the same test objects.

2. The measures *number of errors*, *constructor distance*, and *dynamic error evaluation* are incorporated into the objective functions. A test object has been chosen for which test programs are hard to find. The results are compared to tests where the objective functions incorporated only a boolean measure. This experiment should demonstrate that the application of a more sophisticated measure is indispensable for a successful evolutionary search.

The following settings of the evolutionary algorithm apply for all experiments: 4 subpopulations, 50 individuals per subpopulation, proportional fitness assignment, stochastic universal sampling, discrete recombination, elitest reinsertion with generation gap of 90%.

5.1 Applicability of the approach

In order to demonstrate the applicability of the approach, both an evolutionary test and a random test were performed for the same test cluster. The test cluster used is shown in figure 4. The test goal in line 25 is hard to achieve since for this, three `StateCounter` instances must be created and put into the proper states. The counter values are only affected by a call to `incr()` or `decr()` when the counter object is in state `active`. The evolutionary search was configured to

```
 1: class StateCounter extends Counter
 2: {
 3:   boolean active = false;
 4:   public StateCounter() { counter = 0; }
 5:   public void incr() {
 6:     if( active ) super.incr();
 7:   }
 8:   public void decr() throws Exception {
 9:     if( active ) super.decr();
10:   }
11:   public void activate() {
12:     active = true;
13:   }
14:   public void deactivate() {
15:     active = false;
16:   }
17: }
18: class UseCounter
19: {
20:   public void mut(StateCounter c1,
21:    StateCounter c2, StateCounter c3) {
22:     if( c1.getValue() == 1 )
23:       if( c2.getValue() == 2 )
24:         if( c3.getValue() == 3 )
25:           ;// test goal to be reached
26:   }
27: }
```

Figure 4: test cluster for experiment 1

terminate at least after 50 generations. The maximum test program length was set to 20 statements (the optimum test program without unnecessary statements has the length 11). The measure *dynamic error evaluation* was used for the inconvertible individuals when carrying out evolutionary testing. The random test was executed with the same parameters but without performing the fitness assignment and by using special random operators for recombination and mutation. Overall, eight test runs were carried out. A representative run has been selected for discussion. Figure 5 shows the development of the relative frequency of inconvertible individuals. It indicates that an optimization was necessary in order to find a solution (in generation 14 for evolutionary testing, the random test did not find a solution). Although the random test generated (very few) convertible individuals, this was not sufficient to find the optimum. With evolutionary testing, the number of inconvertible individuals decreased constantly over the generations. Thus, the optimum could be found due to the high number of candidate solutions. The strategy of the two-level optimization proved successful for this experiment.

5.2 Application of the sequence failure measures

The second experiment was carried out to demonstate that it is necessary to use a sophisticated measure for individual evalution in order to find convertible genotypes. In addition, the three suggested measures (see 4.1) have been compared to each other.

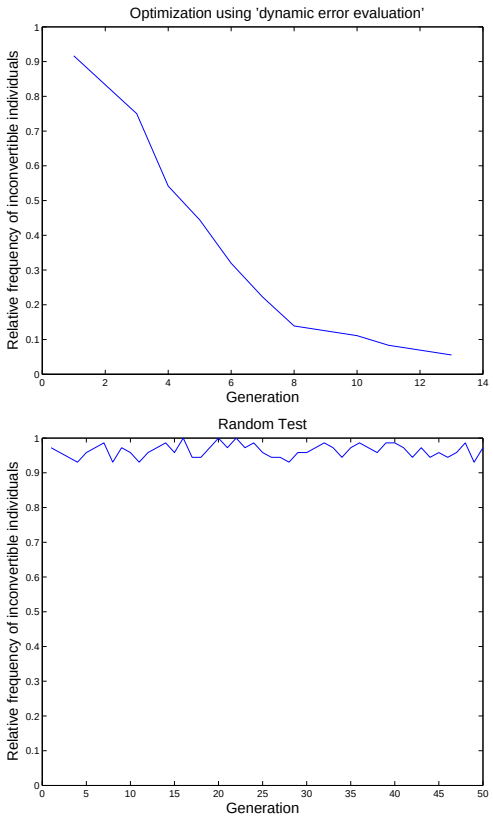

Figure 5: results of comparison with random test

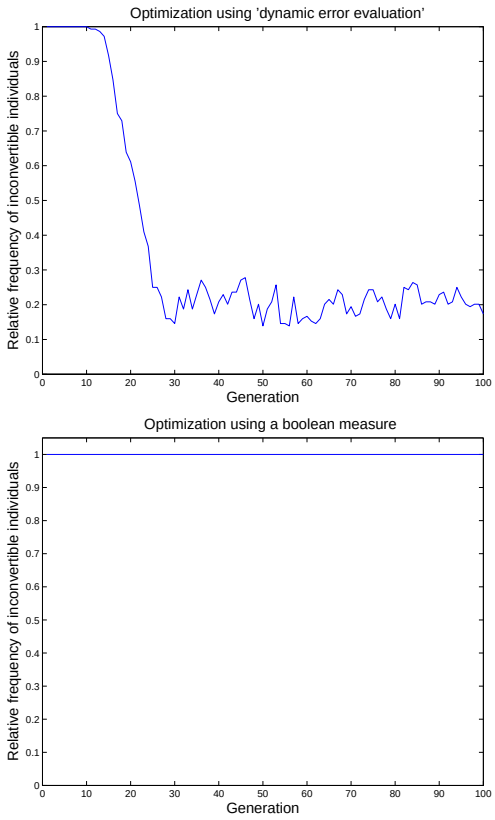

Figure 6: comparison of failure measures

The experimental test cluster consisted of the classes `BinaryTree`, `java.lang.Integer` and `java.lang.Double`. The standard Java classes possess a lot of methods requiring other instances as parameters. Hence, it is hard to find a faultless method call sequence. Ten experiments were carried out for which a boolean failure measure was used that assigned each inconvertible individual the same (non-optimum) objective value $\Omega_I = 20$. Another ten experiments were carried out for each of the three suggested failure measures. Thereby, simplified objective functions were used which assigned each convertible individual the same constant non-optimum value $\Omega_C = 1$ in order to visualize the development of inconvertible individuals over more generations (100 generation was configured as termination criteria). No coverage measurement was carried out. The test was repeated for ten times, a representative run has been selected for discussion. The results in figure 6 indicate that with the boolean measure, the evolutionary algorithm was not able to find convertible individuals. In contrast, using the measure `dynamic error evaluation`, the algorithm was able to generate convertible individuals after 10 unsuccessful generations. The results for the measures *number of failures* and *constructor distance* do not differ essentially from the results of *dynamic error evaluation*. Due to the simplified objective functions, for each measure 100 generations were produced. The relative frequency of inconvertible measures levels off at approximately 20% which indicates that the diversity of a generation is preserved by the evolutionary algorithm. Overall, the application of a sophisticated measure is of high importance for a successful evolutionary test.

6. RELATED WORK

In the area of the evolutionary white-box testing of object-oriented software, only one piece of research is known to the authors, completed by Tonella [8]. Tonella uses evolutionary algorithms to automatically generate unit test classes for given classes. He defined an original evolutionary algorithm with special evolutionary operators for recombination and mutation on a statement level (i.e. his mutation operators, for instance, insert or remove methods from a test program). In his experiments, he generated unit test classes for six Java (JDK) classes.

The limitations of Tonella's approach are the following: using his approach, universal evolutionary algorithms cannot be applied. Additionally, due to the objective functions Tonella uses, the evolutionary search is reduced to a random search in case of complex conditions within the source code which must be satisfied in order to reach a particular test goal. In contrast to Tonella's approach, the strategy presented in this paper makes it possible to employ universal evolutionary algorithms. This means that the suggested encoding allows for the application of other search-based optimization techniques such as hill climbing or simulated annealing as well and to effortlessly change the used strategy. Additionally, using objective functions based on the distance-oriented approach guides the evolutionary search in cases of conditions which are hard to meet by random.

7. FUTURE WORK AND CONCLUSION

As already mentioned in section 4.2, objects represent state machines and some code elements can only be reached when particular objects which are involved in the test scenario are in a certain state, the state behavior must be taken into account accessorily. The chaining approach [2] can be applied as suggested by McMinn and Holcombe [6] in order to identify the methods whose invocation facilitates the achievement of a state-dependent test goal. This approach can be adapted in such a manner that test programs by which these facilitating methods are called, receive a better objective value than programs which do not include them.

Figure 7 shows a class which should be tested. In order to do so, test cases satisfying statement coverage should be generated. Each statement of each method is a test goal for which a test program must be found. Only the statement in line 9 is hard to achieve: the test program must call `decr()` at least ten times for the same object such that the exception will be thrown. Using only the approximation level and distance measures, no guidance is provided to the evolutionary algorithm in this case. When applying the chaining approach, the methods `incr()` and `decr()` are identified as facilitating the change of the variable value `counter`, resulting in the execution of the `if` branch.

```
 1: class Counter
 2: {
 3:   int counter = 10;
 4:   public void incr() {
 5:     counter++;
 6:   }
 7:   public void decr() throws Exception {
 8:     if( isInvalid() )
 9:       throw new InvalidOperationExc();
10:     else counter--;
11:   }
12:   public void getValue() {
13:     return counter;
14:   }
15:   private boolean isInvalid() {
16:     if( counter < 0 ) return true;
17:     return false;
18: }
```

Figure 7: sample class

Nevertheless, the chaining approach must be modified in order to also take into account the *object context* of the events in the event sequence. For each event, it must be defined to which object this event is assigned. Consequently, we propose extending the chaining approach in such a way that an event is not only a tuple $e_i =< n_i, C_i >$ whereby n_i is the ith problem node and C_i the constraint set assigned to that node, but rather a triple

$$e_i =< n_i, C_i, o_i >$$

with o_i as the object context (an object identifier) of the ith problem node in which the event must take place.

The event sequences of a chaining tree can be used to evaluate the method call sequence of a test program. Call sequences with a high coverage of an event sequence are evaluated better than those with a low coverage. Alternatively, the event sequences can be used to create test program skeletons into which an evolutionary algorithm can insert additional statements in order to produce executable and various test programs.

Test cases for testing object-oriented software include test programs which create and manipulate objects in order to achieve a certain test goal. The approach described in this paper facilitates the automatic generation of object-oriented test programs using evolutionary algorithms. The encoding and decoding of test programs into evolvable data structures were discussed. Ideas for the design of objective functions which also consider inconvertible genotype individuals have been presented. In order to demonstrate the its feasability, the approach has been implemented in a prototypic testing system that supports coverage-oriented testing. In the case study, evolutionary algorithms could be employed successfully for the generation of test cases for object-oriented software. The application of the two-level optimization proved successful in all experiments. The results achieved by the experiments are promising and encourage further research in this area. However, the results are preliminary and a lot of more experiments must be carried out. The modification of the chaining approach, in particular, will be investigated in detail in ongoing research. The optimization alternatives (one-level, two-level, three-level) will be compared in order to guide further research and to more thoroughly investigate multi-level optimization.

8. REFERENCES

[1] A. Baresel, H. Sthamer, and M. Schmidt. Fitness function design to improve evolutionary structural testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pages 1329–1336, July 2002. 9-13th July.

[2] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, January 1996.

[3] Genetic and Evolutionary Algorithm Toolbox for use with Matlab. http://www.geatbx.com.

[4] S. Kim, J. A. Clark, and J. A. McDermid. Investigating the applicability of traditional test adequacy criteria for object-oriented programs. In *Proceedings of the ObjectDays 2000*, October 2000.

[5] P. McMinn. Search-based test data generation: A survey. *Journal on Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[6] P. McMinn and M. Holcombe. Hybridizing evolutionary testing with the chaining approach. *Genetic and Evolutionary Computation Conference (GECCO)*, pages 1363–1374, June 2004. June 26-30.

[7] H. Sthamer, J. Wegener, and A. Baresel. Using evolutionary testing to improve efficiency and quality in software testing. In *Proceedings of the 2nd Asia-Pacific Conference on Software Testing Analysis and Review (AsiaSTAR)*, July 2002. 22-24th July.

[8] P. Tonella. Evolutionary testing of classes. *International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–128, 2004. July 11-14.

[9] S. Wappler. Using evolutionary algorithms for the test of object-oriented systems. Master's thesis, Hasso-Plattner-Institute for Software Systems Engineering at University of Potsdam, September 2004.