2006 IEEE Congress on Evolutionary Computation
Sheraton Vancouver Wall Centre Hotel, Vancouver, BC, Canada
July 16-21, 2006

Evolutionary Unit Testing Of Object-Oriented Software Using A Hybrid Evolutionary Algorithm

Stefan Wappler and Joachim Wegener

Abstract— **Evolutionary algorithms have been successfully applied in the area of software testing. However, previous approaches in the area of object-oriented testing are limited in terms of test case feasibility due to call dependences and runtime exceptions. In this paper, we present a search-based approach to automatically generating test cases for object-oriented software. It relies on a tree-based representation of method call sequences. Strongly-typed genetic programming is employed to generate method call trees which respect the call dependences among the methods. We apply a new kind of distance-based fitness function that accounts for runtime exceptions. In a case study, the approach outperformed random testing in terms of achieved coverage and it produced test cases achieving full branch coverage for a test object that makes ample use of explicit runtime exceptions.**

I. Introduction

The application of evolutionary algorithms to the area of software testing has been of increasing interest to many researchers over the recent past years. Evolutionary Testing (ET) which aims at generating relevant test cases for a given software unit by means of an evolutionary algorithm, has been shown to be successful for different test objectives, such as structural testing or temporal testing (e.g. [5]). However, previous research has mainly concentrated on ET approaches for procedural software. In consideration of the growing importance of object orientation in present software development, it is of particular interest to investigate the applicability of evolutionary algorithms to the automatic generation of object-oriented test cases from which software development projects would benefit in terms of increased efficiency and effectivity and hence of reduced development costs.

This paper presents an approach to automatically generating relevant unit test cases for object-oriented software by means of a hybrid evolutionary algorithm. The approach applies a fitness function that accounts for runtime exception which can occur when a candidate test case is evaluated. A unit test case consists of a method call sequence and a number of evaluation checks. The purpose of the method call sequence is to use the given unit under test (typically a class) in a particular scenario. After the method call sequence has been executed, the evaluation checks examine whether the system is in a valid state according to the test reference (e.g. the specification). We propose a tree-based representation of method call sequences which a strongly-typed genetic programming algorithm is able to evolve using well-established evolutionary operators. During fitness evaluation of a method call sequence, we apply a genetic algorithm that modifies the method call sequence in terms of parameter objects and numeric parameter values. Furthermore, we introduce a new distance metric based on method call sequences in order to calculate the fitness of method call sequences that produce runtime exceptions when being evaluated. For reasons of simplicity and without loss of generality, we refer to Java programming and focus on branch coverage for test goal definition. We demonstrate the efficacy of our approach in a case study using a test environment which allows for the complete automation of the entire test case generation process.

The paper is organized as follows: section II outlines unit testing of object-oriented software and describes the nature of object-oriented test cases. Section III explains why runtime exceptions are problematic for fitness calculation. Section IV presents our approach based on strongly-typed genetic programming and a fitness function based on a hierarchical distance metric. Section V contains a case study that we performed for evaluating the approach. Finally, section VI concludes the paper and gives some directions for further research.

II. Unit Testing of Object-Oriented Software

Unit testing is an activity that is usually performed in early development stages in order to detect errors in the software and to gain confidence in the correctness of the software if no errors are found. In the context of object orientation, a particular class which is part of the application to be developed is considered to be a single unit for the test. The idea of unit testing is to use the unit under test (UUT) in interesting scenarios in order to detect erroneous behavior. Since it is usually impossible to exercise the UUT in all conceivable scenarios, an adequacy criterion such as branch coverage or function coverage is used that decides which scenarios are relevant. Indirectly, such a criterion is used as a termination criterion for the process of test case generation: if the entirety of test cases generated so far satisfies the adequacy criterion, no more test cases need to be created.

Usually, the test of a single class involves the usage of other classes, too. For instance, classes that appear in the signatures of the methods of the class under test (CUT) are required for the test of this class. The transitive set of classes which are relevant for testing the CUT is called *test cluster* for the CUT.

Stefan Wappler is with the Technical University of Berlin, Daimler-Chrysler Automotive IT Institute, Ernst-Reuter-Platz 7, D-10587 Berlin, Germany, phone: +49-30-39982-358, email: stefan.wappler@tu-berlin.de

Joachim Wegener is with DaimlerChrysler AG, Research and Technology, Alt-Moabit 96a, D-10558 Berlin, Germany, phone: +49-30-39982-232, email: joachim.wegener@daimlerchrysler.com

0-7803-9487-9/06/$20.00/©2006 IEEE

Essentially, a unit test case for object-oriented software comprises a method call sequence and one or more assertion statements. The method call sequence (also referred to as *test program*) represents a particular test scenario in which objects (instances of the test cluster classes) that are needed for the test are created and put into a particular state by calling several instance methods for these objects. After the execution of the method call sequence the assertion statements check whether the system is in the expected state. The use of a coverage-oriented adequacy criterion for testing an object-oriented class requires test cases to be generated that satisfy the criterion for each method of this class. Consequently, a test case usually focuses on one particular method, the *method under test* (MUT). Figure 1 shows the

```
class Controller
{
  public Controller(Config cfg)
  public void reconfigure(Config cfg)
  public Config getConfig()
  public void connect()
  public int retrieve(int signal)
  public void disconnect()
}

class Config
{
  public Config(int port, int count)
  public int getPort()
  public int getSignalCount()
}
```

Fig. 1. test cluster for class `Controller`

test cluster for class `Controller` which will be assumed to be the CUT from now on (only the public interface is shown). The test cluster consists of class `Controller` and class `Config`. Class `Config` is used as a parameter type of the methods `Controller(Config)` and `Controller.reconfigure(Config)` and is hence required for the test. Figure 2 shows a test case that focuses

```
// test scenario
Config cfg1 = new Config( 0x0A, 5 );
Config cfg2 = new Config( 0x0B, 2 );
Controller ctl = new Controller( cfg1 )
ctl.reconfigure( cfg2 );

// test evaluation
assert(
  ctl.getConfig().getPort() == cfg2.getPort() );
assert(
  ctl.getConfig().getSignalCount() ==
  cfg2.getSignalCount() );
```

Fig. 2. example test case for method `A.ma2(B)`

on the method `Controller.reconfigure`. The test scenario consists of the creation of two instances of class `Config` and one instance of class `Controller` where one `Config` instance is used as the parameter object. Afterwards, method `Controller.reconfigure(Config)` is called with the other instance of `Config`. Finally, the test case checks whether the configuration of the controller is as expected. If it is, the test passes, otherwise it fails.

III. Runtime Exceptions

In the context of Java programming, there are different kinds of situations where exceptions occur. Exceptions serve to indicate that the normal flow of control encounters a state which makes it impossible to continue processing normally. Additionally, exceptions provide the possibility of reacting to these states appropriately. Many programming languages offer the ability to explicitly produce a runtime exception, e.g. in Java via a `throw` statement. Making use of this ability is for instance reasonable in cases where a given file name identifies a file that does not exist or where a provided argument has an invalid value. Alternatively, some types of exceptions are produced implicitly by the runtime system. Such exceptions occur for instance during attempts to call a method for a `null` object reference or attempts to access an array position with an index that is larger than the array size.

When using the goal-oriented approach in the context of branch coverage, each branch of the class's control flow graphs becomes an individual test goal for which a test case is searched. During the evolutionary search for a test case that leads to the achievement of the current test goal (in our case an individual branch of the class under test), a runtime exception can occur that prematurely terminates the execution of the current method call sequence. An exception can occur in the following contexts:

1) The current target branch leads to an explicit throw statement; this throw statement is reached and the corresponding exception is thrown.
2) An exception occurs before the current target branch can be traversed (regardless of whether or not the current target branch leads to a throw statement).

While case 1 is unproblematic (a test case has been found for the current test goal), case 2 is critical with respect to fitness calculation. In the following, we will focus on the second case.

```
class Controller
{ ...
  Signal[] signals;

  public Controller(Config cfg) {
    int sigCount = cfg.getSignalCount()
    signals = new Signal[sigCount];
    ...
  }

  public int retrieve(int signal) {
    if( signal < 0 || signal-1 > signals.length )
      throw new IllegalArgumentException();
    return signals[signal].value();
  }
}
```

Fig. 3. code fragment for class `Controller`

Figure 3 shows a fragment of the source code of the `Controller` class. The class encapsulates an array containing a number of `Signal` objects. The constructor `Controller(Config)` initializes the size of this array with the number of signals given by the `Config`

object passed as argument to the constructor. Method `Controller.retrieve(int)` serves to query the value of a signal using the index `signal` to identify the relevant signal. In case of an invalid signal index, this method throws a runtime exception. Note that an exception would also be thrown even if the argument-checking code and the explicit throw clause were not present; this code makes the issue even more obvious (without the checking code, an "array index out of bounds exception" would be produced instead of an "illegal argument exception"). The test case shown in figure 4 shows a feasible

```
// generated test scenario
Config cfg1 = new Config( 2103, 5 );
Config cfg2 = new Config( 31234, 132 );
Controller ctrl1 = new Controller( cfg1 );
ctrl1.retrieve( -173 );
ctrl1.reconfigure( cfg2 );
```

Fig. 4. test case for method `Controller.reconfigure(Config)`

method call sequence which aims at executing the method `Controller.reconfigure(Config)`. Although it is feasible, an exception indicating that an invalid array access has occurred will be produced when this sequence is executed. This is due to the parameter value for `signal` when calling method `Controller.retrieve(int)`. This value should be in the range of the signal array size. In the case of figure 4, a valid index would be in the range of $[0, 4]$. The exception leads to a premature termination of the execution of the method call sequence. Consequently, the current method under test is not executed.

The previous approaches in the area of object-oriented evolutionary testing [7], [8], [3] make use of distance metrics or coverage metrics based on monitored execution information on the method under test when calculating the fitness of a method call sequence. In the case of runtime exceptions – if they do not belong to the current target or test goal of the search – no adequate fitness value can be computed that directs the search suitably. This is due to the fact that no monitoring information will be created when the method under test is not executed. Consequently, either the complete approach fails in this case, or the search degenerates to a random search if a constant penalty is used as the fitness value for exceptional sequences.

IV. Our Approach

This section outlines the hybrid evolutionary algorithm that we use to generate test cases (section IV-A), presents a tree-based representation of object-oriented unit test cases (section IV-B), and describes in more detail the fitness function that we apply (section IV-D).

A. Evolutionary Algorithm Design

As we will describe in section IV-B, we use trees in order to represent method call sequences. These trees contain the information of which methods should be called in sequence, and which target objects and parameter objects should be used for the individual method calls. Since the tree representation does not support the reuse of objects for multiple method calls, we must optimize the object assignments during an additional optimization step. Additionally, numeric basic type parameters are also optimized during this additional optimization step. Hence, before evaluating a tree individual, an evolutionary search is performed in order to improve it and to acquire favorable parameter values. Figure 5 outlines the test-case-generating evolutionary algorithm[1]. Initially, a population of tree individuals is

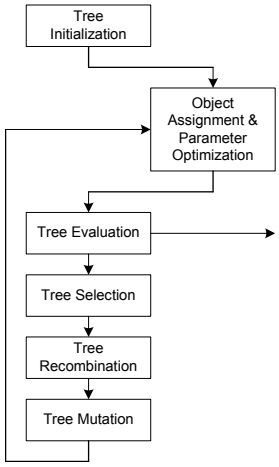

Fig. 5. algorithm workflow

generated by random. For each individual, an evolutionary search is performed to "fine-tune" the object assignments and to generate numeric parameter values. The evaluation of a tree individual consists of assigning the fitness of the best individual of the parameter and object reference optimization to the original tree individual. If an ideal individual is found or another termination criterion applies, the search terminates after fitness evaluation. Otherwise, promising tree individuals are selected, recombined and mutated in order to create preferably better ones.

B. Method Call Sequence Generation using strongly-typed genetic programming

In section II, we described the structure of an object-oriented test case. More formally, such a test case can be understood as being a sequence $S =< m_1, m_2, ..., m_n >$ of method calls $m_i \in M$ where M is the set of all public methods of the test cluster classes. Typically, the methods m_1 to m_{n-1} create the objects that participate in the test and put them into particular states. Finally, m_n is the call of the method under test. A method m_i can only be executed if an appropriate target object and all required parameter objects for this call have been created in advance, i.e. during the calls of m_1 to m_{i-1}. Obviously, call dependences exist among the methods of M which must be taken into account when

[1] In [9], we considered the algorithm as being a two-level evolutionary algorithm. The interpretation of the algorithm as a hybrid algorithm emphasizes more on the nature of the parameter and object assignment search as some kind of lifetime learning (see VI).

generating method call sequences. This also means that not every arbitrary call sequence is feasible – only those which regard all call dependences.

As described in more detail in [9], we use method call trees for the representation of method call sequences in order to ensure feasibility of the generated method call sequences. These method call trees are acyclic directed subgraphs of the *extended method call dependence graph* (EMCDG) of the given test cluster. The EMCDG is a bipartite directed multigraph which models the call dependences that exist among the methods of the test cluster classes. Nodes of type 1 (method nodes) represent the methods, whereas nodes of type 2 (class nodes) represent the classes of the test cluster. A link from a method node to a class node indicates that the method can only be called if an instance of the class represented by that class node is available. A link from a class node to a method node means that the method represented by the connected method node delivers an instance of the class. The graph also assumes that each instance method of a class is indirectly able to deliver an instance of that class – namely the target object used by this method. Figure 6 shows the EMCDG for the test cluster of figure 1. For example, if we want to call method `Controller.reconfigure(Config)`, the graph indicates that we need to have an instance of class `Controller` (used as target object), as well as an instance of class `Config` (used as parameter object). The dependence paths along the nodes, including the decisions actually made at the class nodes as to which method should be used for instance delivering, are represented by the method call trees. A method call tree that results in calling method `Controller.reconfigure(Config)` is shown in figure 7.

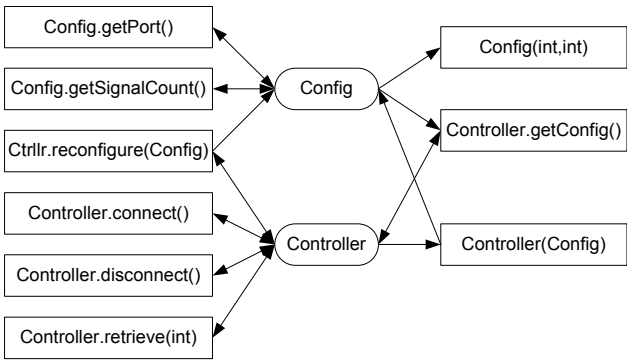

Fig. 6. extended method call dependence graph

In order to employ strongly-typed genetic programming (STGP), the function set and the type set must be defined with respect to the test object at hand. The methods of the test cluster classes constitute the function set. Each method becomes an STGP function and is added to the function set in the following way:

- The return type of the STGP function is defined as the class to which the method belongs.
- The first child type of the STGP function is defined as the class to which the method belongs, unless it is a constructor or static method.
- Objective parameters of the method are defined as additional child types of the STGP function.
- If the method returns an object, the method is inserted twice into the function set, and the return type of the second instance is defined as the actual return type of the method.

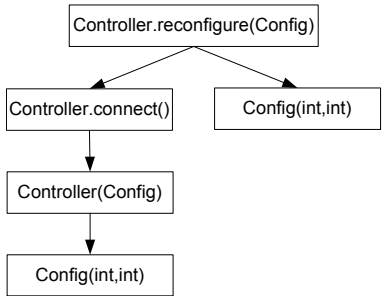

Fig. 7. example call tree for `Controller.reconfigure(Config)`

The tree itself is typed using the particular type τ. Since a call sequence aims at finally invoking the method under test, the method under test is added to the function set again with τ defined as the return type. This ensures that all generated trees call the method under test at least once. Defining the return type of the STGP functions as the class to which the corresponding method belongs allows method calls to be "concatenated", and therefore makes it possible to have multiple method invocations for the same object.

In order to take into account the polymorphic relationships that exist due to inheritance relations, the STGP types are specified in a way that reflects the type hierarchy of the test cluster classes. We use set-based typing as implemented by Luke [11]. A set type is a type identifier which is assigned a set of type identifiers. Two types are considered compatible if their type set intersection is non-empty. The overall type set of an STGP problem consists of the atomic type identifiers and the set types. Using set-based typing supports multiple inheritance.

A method call tree is transformed into a call sequence using in-order linearization. Figure 8 shows how the linearized method call tree of figure 7 would look. In this sequence, the types and automatically generated variable names have already been inserted. In some cases, multiple instances are available which can serve as objects for a particular method call. For instance, `config1` could have been used as the parameter object for the last method call of the example sequence (parameter object in italic) instead of `config2`. This "degree of freedom" with respect to the

```
Config config1 = new Config(int,int);
Controller controller1 = new Controller(config1);
controller1.connect();
Config config2 = new Config(int,int);
controller1.reconfigure(config2);
```

Fig. 8. linearized method call tree

object assignments is dealt with during the object assignment and parameter search. It is necessary to allow for object reuse since the tree-based representation does not support object reuse per se.

C. Object Assignment and Parameter Search

A set of variables can be derived from each method call tree that represents the object assignments and numeric parameter values. These variables are optimized using a genetic algorithm working on double vectors. For the example of figure 8, the following variable vector V would be optimized:

$$V = (int, int, int, int, \{1, 2\})$$

where `int` is the range of signed integer values and the set $\{1, 2\}$ serves to select between the two candidate parameter objects `config1` and `config2`.

D. Fitness Evaluation

We employ a distance-based fitness function which expresses how *close* execution of a test program is to reaching the current test goal (the program element to be covered). This closeness is expressed in terms of the three distances *method call distance* d_{MC}, *control node distance* d_{CN}, and *local problem node distance* d_{PN}. The overall fitness f of a test program is the sum of these appropriately weighted[2] distances:

$$f = \lambda d_{MC} + d_{CN} + d_{PN} \quad (1)$$

$f \geq 0$ is a minimizing fitness function, 0 is the optimum. The weight $\lambda = |P_{max}| + 1$ scales the method call distance appropriately. $|P_{max}|$ is the number of control nodes of the longest loop-free method control flow graph path of the test cluster's methods. The usage of λ ensures that $\lambda d_{MC}^{min} > d_{CN}^{max}$ and $d_{CN}^{min} \geq d_{PN}^{max}$ holds where d_{MC}^{min} is the smallest non-null method call distance, d_{CN}^{max} is the greatest control node distance, d_{CN}^{min} is the smallest non-null control node distance, and d_{PN}^{max} is the greatest problem node distance. The metrics take into account runtime exceptions. In order to do so, each works on a different level of granularity:

1) d_{MC} on the *method call level*
2) d_{CN} on the *method control flow graph level*
3) d_{CN} on the *problem node level*

In the following, we will explain the distance metrics for these three levels with the help of the example test case shown in figure 4.

On the method call level, the distance metric d_{MC} expresses how close execution approached the method under test in terms of number of methods. In case of a runtime exception, execution of a method call sequence terminates prematurely, meaning that the method under test is not called. The number of methods which were not executed due to an exception is the method call distance d_{MC}. Let $S =< m_1, m_2, ..., m_l >$ be a method call sequence, $l = |S|$ be the length of the sequence, and $e \in \{1, 2, ..., l\}$ be the index of the method at which execution stops (either $e = l$ or m_e produced an exception). Then d_{MC} is defined as follows:

$$d_{MC} = l - e \quad (2)$$

For the example test case (figure 4), the value of d_{MC} would be 1 since $l = 5$ and $e = 4$.

On the method control flow level, the distance metric d_{CN} expresses how close execution approached the target node in terms of the number of control graph nodes of the method that was called last. The distance between the closest approach of the execution path and the target node is measured. Figure 9 shows the control flow graph of

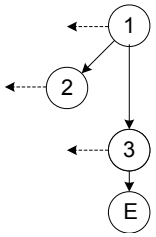

Fig. 9. control flow graph of method `Controller.retrieve(int)`

method `Controller.retrieve(int)`. E denotes the target node to be reached[3] (in our case, E is the exit node of the control flow graph). A branch is said to be *critical* if it is impossible to reach the target once the control flow has diverged down that branch. The node at which execution takes a critical branch is referred to as the *problem node*. In principle, each statement represented by a control node can produce an exception[4]. Thus, each control node possesses an additional critical branch (referred to as *exceptional branch*), as indicated by the dashed arrows coming from each control node. The distance d_{CN} corresponds to the metric *approximation level* ([10]). The value of d_{CN} is equal to the number of nodes that belong to the longest path that starts at the problem node and aims at the target node (excluding the target node). According the the example test case, $P_E =< 1, 2 >$ is the execution path with respect to method `Controller.reconfigure(int)`. It contains the control nodes 1 and 2. It turns out that node 1 is the problem node since from there execution diverged along a critical branch. The longest path from the problem node 1 to the target node E is the path $P^*_{PN \to E} =< 1, 3 >$ for our example; hence $d_{CN} = 1$. The distance d_{CN} is defined as follows:

$$d_{CN} = |P^*_{PN \to T}| \quad (3)$$

where T is the current target node. If the considered control flow graph belongs to the method under test, the target node

[2]Since the fitness function returns exactly one real value, the individual distances must be integrated into one value.

[3]When using branch coverage, empty statements are inserted for the complementary branches of a condition if there is only one branch present in the code. Consequently, instead of the branches of the control flow graph, the control nodes are used as test goals.

[4]We assume this for reasons of simplicity. A static analysis can be carried out in order to remove all non-branching *safe* nodes from the control flow graph.

T is the current test goal. Otherwise, T is the exit node of the method at which execution terminated. In this case, d_{CN} expresses how close execution is from properly returning from the current method call.

On the problem node level, the distance d_{PN} expresses how far execution is away from diverging along the branch of the problem node which leads to the target. There are two reasons why an undesired branch is taken:

- either the problem node is a branching node and the condition of this node is evaluated unfavorably; hence, the critical (not exceptional) branch is taken. Then, d_{PN} is proportional to the conditional distance d_{LD} of the problem node's predicate according to the distance functions used for testing procedural software ([10]). For instance, the distance of an equality checking condition `if( a == b )` is $d_{=} = (1+\varepsilon)^{-|a-b|}$ with $\varepsilon \in (0,1)$.
- or execution from the problem node has diverged along an exceptional branch. Since it is essentially unclear how close most of the exceptions are from not being raised, d_{PN} is assigned the constant value 2 in this case.

Altogether, the problem node distance can be formulated as follows:

$$d_{PN} = \begin{cases} \frac{1}{2} d_{LD} & \text{critical branch taken is no exceptional branch} \\ 1 & \text{otherwise} \end{cases} \quad (4)$$

where $d_{PN} \in [0,1]$ and $d_{LD} \in [0,1]$. The factor $\frac{1}{2}$ in front of d_{LD} is necessary to distinguish between the case that either a condition has been evaluated unfavorably with maximum distance or that an exception occurred. For *gradual exceptions* – exceptions for which a gradual distance can also be calculated such "as array index out of bounds exceptions" – we use the gradual distance information instead of the constant 1. This exceptional distance is mapped into the interval $(\frac{1}{2}, 1]$.

V. Case Study

In this section, we describe a case study which consists of a comparison of our evolutionary approach to random testing. The comparison to random testing aims at demonstrating that an optimization is required for the generation of test cases for some of the test goals. At first, we describe our test environment used for experimentation. Afterwards, the considered test object is characterized. Finally, the achieved results are presented and discussed.

A. Test Environment

The approach presented in this paper was implemented in a Java-based tool which makes use of third party toolboxes for genetic programming and evolutionary algorithms. Off-the-shelf genetic programming algorithms and structures were used from the ECJ (Evolutionary Computation for Java) system developed by Luke [4], [11]. Object assignment and parameter optimization employed the Genetic and Evolutionary Algorithms Toolbox (GEATbx) provided by Pohlheim [2]. The OpenJava framework [6] was used to parse and instrument Java source code. A special OpenJava metaclass has been implemented to enable branch coverage instrumentation.

Figure 10 shows a high-level overview of the test case generator. Rectangular nodes represent active components, while rounded nodes represent passive components (storages and channels). The source code of the test cluster classes is parsed

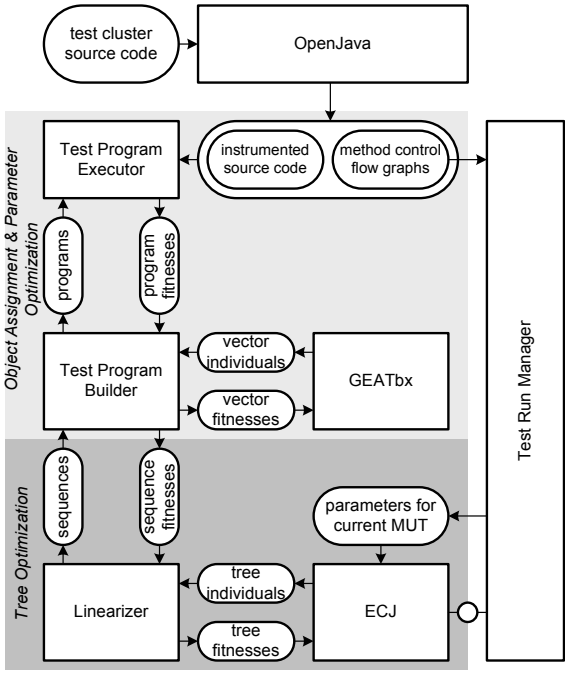

Fig. 10. test case generator system overview

and instrumented by the OpenJava framework. The control flow graphs of the particular methods are constructed during parsing, and serve as the basis for test goal definition. Hence, the outcome of this preparation step is an instrumented version of the classes and the method control flow graphs. The Test Run Manager (TRMan) uses the graphs of the CUT to derive the individual test goals. TRMan generates an ECJ parameter file for each test goal and calls the ECJ system which starts a GP optimization. The parameter file contains the function set definition, the type set definiton, and general optimization settings such as evolutionary operator selection. The Linearizer takes the trees produced by ECJ, linearizes them (as described in section IV-B) and hands them over to the Test Program Builder (TPB). TPB identifies the optimizable object assignments and parameters, and generates the corresponding variable specification. Object assignment and parameter search is carried out using GEATbx. The method call sequence is modified according the values that this toolbox delivers. The resulting programs are evaluated by the Test Program Executor (TPE). TPE executes the programs using the instrumented classes and monitors the execution flow. It calculates the distance metrics (section IV-D) and delivers the fitnesses back to TPB which forwards them to GEATbx. TPB returns the overall best fitness achieved during

object assignment and parameter search as the fitness of the tree individual for which this search was carried out.

B. Test Object

The system described above has been used to carry out a case study demonstrating the feasibility and efficacy of the approach, particularly with respect to runtime exceptions.

```
public class Controller {
    protected final static int MAX_SIGNALS = 5;
    protected final static int MIN_PORT    = 8000;
    protected final static int MAX_PORT    = 8005;
    private Config             cfg         = null;
    private int[]              signals     = null;

    public Controller() {
        cfg = new Config( -1 );
        signals = new int[cfg.getSignalCount()];
    }
    public void reconfigure(Config cfg) throws Exception {
        if( cfg.getSignalCount() > MAX_SIGNALS )
/*t1*/      throw new Exception("Too many signals.");
        if( cfg.getPort()<MIN_PORT || cfg.getPort()>MAX_PORT )
            throw new Exception("Invalid port.");
/*t2*/  this.cfg = cfg;
        signals = new int[cfg.getSignalCount()];
    }
    public int retrieve(int signal) {
      if( signal<0 || signal>signals.length-1 )
        throw new IllegalArgumentException("Invalid signal.");
      return signals[signal]; }
    public Config getConfig() { return cfg; }
}
```

Fig. 11. source of class `Controller`

```
public class Config {
    private Vector          signals;
    private int             port;

    public Config(int port) {
        this.port = port;
        signals = new Vector();
        // base signal (clock)
        addSignal( 0 );
    }
    public void addSignal(int signalType) {
        signals.add( new Integer( signalType ) );
    }
    public int getSignalCount() { return signals.size(); }

    public int getPort() { return port; }

    public void setPort(int port) { this.port = port; }
}
```

Fig. 12. source of class `Config`

Figure 11 shows the source code of the class under test – class `Controller` – and figure 12 shows the code of class `Config`. Both classes constitute the test cluster.

Method `Controller.reconfigure(Config)` contains two difficult test goals: test goal $t1$ which can only be reached if there are more than 5 signals configured (meaning that the test program must include at least 5 calls to `Config.addSignal(int)`), and test goal $t2$ which can only be reached if the specified port has a valid value (meaning that either the constructor `Config(int)` or the method `Config.setPort(int)` must be called with a suitable parameter).

The following settings were used for the evolutionary algorithm:

- ECJ: 1 subpopulation; 10 individuals per subpopulation; initialization: half/full (17 max. tree depth); selection: tournament; recombination: subtree crossover; mutation: demotion and promotion [1], point mutation; termination: at least after 10 generations;
- GEATbx: 4 subpopulations, 10 individuals per subpopulation; initialization: random values; selection: stochastic universal sampling; recombination: discrete; mutation: real mutation; reinsertion: elitest with generation gap 0.9; termination: at least after 50 generations or if average best fitness over 15 generations does not improve

C. Test Results

We ran the evolutionary test case generator for the given test cluster in order to generate test cases satisfying branch coverage. The set of generated test cases achieved full (100%) branch coverage. During the search, 11966 test programs were generated and evaluated. The resulting test set contained 3 test cases. After that, we configured the evolutionary algorithm to use random operators in order to realize a random test case generator. We used this generator to produce random test cases for the given test cluster. According to the specified termination criteria, it stopped after having evaluated 43233 test programs. In total, the generated test cases achieve a coverage of 66%. No test cases for the test goals $t1$, $t2$ and all dependent test goals were produced.

The difficulty of generating a suitable test case for the test goals does not only depend on the fact that either a special numeric constant must be generated or a particular number of calls to the same method must be made. Rather the search is hindered by exceptions that occur when executing a method sequence containing method calls which raise an exception for many of the values of the parameters' input domains. For example, method `Controller.retrieve(int)` raises an exception if the provided signal index is out of range. This applies for most of randomly generated signal indices. Figure 13 shows a generated test case for testing method `Controller.retrieve(int)` (the parameters to be optimized during parameter search are in italic). The difficulty with this test case is to generate appropriate values for the parameters `param1` and `param2`. In order to cover the else branch of the first condition of `Controller.retrieve(int)`, both values must be 0. Figure 14 illustrates the fitness landscape for this search. Given this landscape, the genetic algorithm is guided to first set the value of `param1` to 0 and afterwards to set the value of `param2` to 0 as well. Consequently, the target branch will be traversed by the corresponding test case.

```
Config config1 = new Config(int,int);
Controller controller1 = new Controller(config1);
controller1.retrieve(param1);
controller1.retrieve(param2);
```

Fig. 13. generated test case for method `Controller.retrieve(int)`

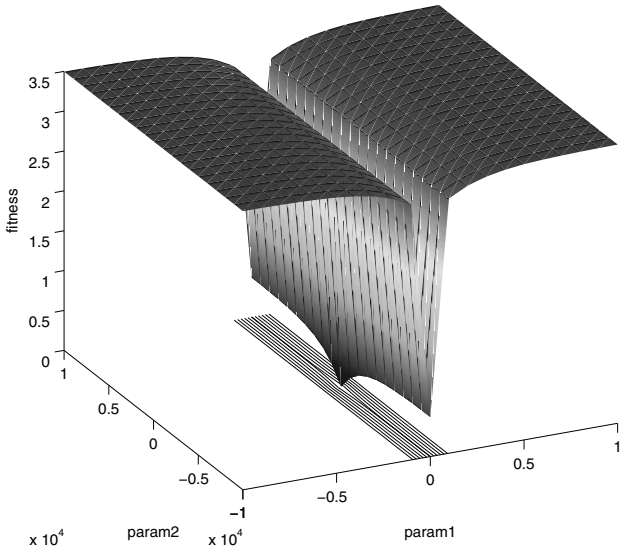

Fig. 14. fitness landscape for the example test case ($\lambda = 3, \varepsilon = 0.0005$)

The results achieved are promising. However, many more experiments must be carried out for a more comprehensive validation.

VI. Conclusion and Future Work

In this paper, we presented an approach using a hybrid evolutionary algorithm for the generation of method call sequences. These sequences form the basis for creating test cases for the unit testing of object-oriented software. The hybrid evolutionary algorithm consists of a strongly-typed genetic programming algorithm and a genetic algorithm for parameter search. By using strongly-typed genetic programming, feasibility of the method call sequences is preserved throughout the entire search process. We described a procedure for how to define the function set and the type set of a standard STGP algorithm. Function set definition is based on the signatures of the methods of the test cluster classes. Type set definition is based on the inheritance relations of the test cluster classes. Polymorphism is supported using set-based types. We defined a distance-based fitness function which takes into account runtime exceptions. This fitness function makes use of a distance metric that is based on the number of unexecuted methods of a method call sequence. Unlike previous approaches in this area, the search is guided well in case of uncaught runtime exceptions. A case study demonstrated the feasibility of the approach. In an experiment, the approach outperformed random testing. In contrast to random testing, full branch coverage could be achieved completely automatically.

Further research particularly needs to investigate the problem node distance functions for operators specific to object-orientation. For instance, the distance functions for object address comparisons and for type checks must be improved. Furthermore, it must be investigated how to deal with runtime exceptions that are required to be raised in order to achieve a subsequent test goal. Additionally, the way of covering non-public methods is an item of consideration for future work. Efficiency of the evolutionary algorithm could be improved by "exchanging" the numeric parameter values among the tree individuals: the numeric values that have been optimized during parameter search can be stored within a tree individual and hence will be available after tree recombination and tree mutation. The parameter search for the offspring tree individuals can use the numeric values as start values.

References

[1] K. Chellapilla. A preliminary investigation into evolving modular programs without subtree crossover. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 23–31, 1998.

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

[3] X. Liu, B. Wang, and H. Liu. Evolutionary search in the context of object-oriented programs. In *MIC2005: The Sixth Metaheuristics International Conference*, September 2005.

[4] S. Luke. *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat*. PhD thesis, Department of Computer Science, University of Maryland, College Park, Maryland, 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] M. Tatsubori, S. Chiba, M. Killijian, and K. Itano. OpenJava: A class-based macro system for Java. *Lecture Notes in Computer Science 1826, Reflection and Software Engineering*, pages 117–133, 2000.

[7] P. Tonella. Evolutionary testing of classes. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 119–128, New York, NY, USA, 2004. ACM Press.

[8] S. Wappler and F. Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1053–1060, New York, NY, USA, 2005. ACM Press.

[9] S. Wappler and J. Wegener. Evolutionary testing of object-oriented software using strongly-typed genetic programming. In *GECCO '06: Proceedings of the 2006 conference on Genetic and evolutionary computation*, New York, NY, USA, 2006. ACM Press.

[10] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(1):841–854, 2001.

[11] G. C. Wilson, A. McIntyre, and M. I. Heywood. Resource review: Three open source systems for evolving programs - lilgp, ecj and grammatical evolution. *Genetic Programming and Evolvable Machines*, 5(1):103–105, 2004.