Evolutionary Unit Testing of Object-Oriented Software Using Strongly-Typed Genetic Programming

Stefan Wappler
Technical University of Berlin
DaimlerChrysler Automotive IT Institute
Ernst-Reuter-Platz 7, 10587 Berlin, Germany
Phone: +49 30 39982 358
stefan.wappler@tu-berlin.de

Joachim Wegener
DaimlerChrysler AG
Research and Technology
Alt-Moabit 96a, D-10559 Berlin, Germany
Phone: +49 30 39982 232
joachim.wegener@daimlerchrysler.com

ABSTRACT

Evolutionary algorithms have successfully been applied to software testing. Not only approaches that search for numeric test data for procedural test objects have been investigated, but also techniques for automatically generating test programs that represent object-oriented unit test cases. Compared to numeric test data, test programs optimized for object-oriented unit testing are more complex. Method call sequences that realize interesting test scenarios must be evolved. An arbitrary method call sequence is not necessarily feasible due to call dependences which exist among the methods that potentially appear in a method call sequence. The approach presented in this paper relies on a tree-based representation of method call sequences by which sequence feasibility is preserved throughout the entire search process. In contrast to other approaches in this area, neither repair of individuals nor penalty mechanisms are required. Strongly-typed genetic programming is employed to generate method call trees. In order to deal with runtime exceptions, we use an extended distance-based fitness function. We performed experiments with four test objects. The initial results are promising: high code coverages were achieved completely automatically for all of the test objects.

Categories and Subject Descriptors

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

General Terms

Verification

Keywords

automated test case generation, evolutionary testing, object-orientation, strongly-typed genetic programming

GECCO'06, July 8–12, 2006, Seattle, Washington, USA.

1. INTRODUCTION

Evolutionary Testing (ET) has been shown to be successful in automatically generating relevant unit test cases for procedural software [7]. Applying ET can increase efficiency and quality of the usually costly test data generation process [9]. The application of search-based strategies for object-oriented unit testing has not yet been investigated comprehensively; only three approaches are known to the authors ([11, 12, 5]).

With ET, the task of test case generation is formulated as a search problem which is tried to be solved using an evolutionary algorithm. Test cases for the unit testing of object-oriented software consist of a sequence of method calls (the *test program*) that realize a particular test scenario, and the test evaluation typically expressed by assertion statements which are evaluated after the test scenario has been executed. In order to apply evolutionary algorithms, a suitable representation of object-oriented test programs must be defined. Furthermore, a fitness function must be formulated which is able to distinguish between interesting test programs and bland ones. The search space of the evolutionary search is the set of all conceivable test programs for a given test object. The fitness function guides the evolutionary search to search space regions that contain interesting test programs for the test objective at hand.

This paper presents an approach which applies strongly-typed genetic programming (STGP) to the generation of object-oriented test programs. Method call sequences are represented by method call trees. These trees are able to express the call dependences of the methods that are relevant for a given test object. Feasibility in terms of regarded call dependences is preserved by all evolutionary operators that work on the method call trees. We apply the approach to generating test cases that satisfy branch coverage of Java classes. Nevertheless, it can be easily adapted to also work for other coverage criteria.

The paper is organized as follows. Section 2 recapitulates unit testing of object-oriented software. Feasibility of method call sequences is dealt with in section 3. Section 4 introduces strongly-typed genetic programming. Section 5 describes our approach based on STGP. The results of experiments performed with the approach are presented in section 6. Finally, section 7 concludes the paper.

2. OBJECT-ORIENTED UNIT TESTING

The primary aim of unit testing is to uncover errors within a given unit (the test object) or, if no errors can be found,

to gain confidence in its correctness. For doing so, the test object is used in different scenarios which are considered to be interesting or relevant for this test object. In the context of object-orientation, the particular classes which constitute an application are considered to be the smallest units that can be tested in isolation. Test set adequacy criteria, such as branch coverage, are used to answer the question of what interesting test scenarios are and when the process of test case generation can be terminated. A test set is said to be adequate with respect to a given criterion if the entirety of test cases in this set satisfies this criterion. For example, a set of test cases that lead to the traversal of all branches of the control flow graphs of the test object when all test cases are executed, is adequate with respect to branch coverage.

Testing a single class usually also involves other classes, e.g. classes that appear as parameter types in the method signatures of the class under test (CUT). The transitive set of classes which are relevant for testing a particular class is called the *test cluster* for this class.

```
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()
}
```

Figure 1: test cluster for class `Controller`

A unit test case for object-oriented software consists of a method call sequence and one or more assertion statements. The method call sequence represents the test scenario. During its execution, all objects participating in the test are created and put into a particular state by calling several instance methods for these objects. The assertion statements check whether the system is in a valid state after the execution of the method call sequence. When testing an object-oriented class using a coverage-oriented adequacy criterion, test cases must be generated which satisfy the criterion for all methods of this class. Hence, each test case focuses on the execution of one particular method, the *method under test* (MUT). Consequently, the entirety of adequate test cases for each method of the class under test satisfies the given adequacy criterion for the whole class.

Figure 1 shows the public interface of a `Controller` class as well as the public interface of a `Config` class, which is used by the `Controller` class (and hence is part of the test cluster). Class `Controller` will be assumed to be the CUT from now on. An example test case for testing method `Controller.reconfigure(Config)` is depicted in figure 2. The test scenario consists of creating two instances of class `Config` with particular parameters and one instance of class `Controller` using one of the `Config`s as parameter. Finally, the test case checks whether the configuration returned by method `Controller.getConfig()` equals the configuration

```
// 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() );
```

Figure 2: example test case for method `Controller.reconfigure(Config)`

which has been previously passed to method `Controller.reconfigure(Config)`. If the checks are successful, the test passes, otherwise it fails.

3. CALL SEQUENCE FEASIBILITY

The application of evolutionary algorithms for the automatic generation of test cases for procedural software has been investigated thoroughly during the last decade, and successful approaches have been developed. Inspired by the approaches for testing procedural software, new search-based approaches for testing object-oriented software have been recently researched (e.g. [11, 12, 5]). With procedural evolutionary testing, a candidate solution is represented by a vector of real or integer values. In principle, no constraints or consistency criteria exist among the variable of such a vector. However, the representation of a candidate solution for object-oriented evolutionary testing must encode the complete method call sequence which consists of arbitrary method calls including the actual parameters. Since call dependences exist among the methods of the test cluster, not any arbitrary method call sequence is executable. Figure 3 shows two method call sequences without actual

```
// infeasible:          // feasible:

Config(int,int);        Config(int,int);
Config.getPort();       Controller(Config);
Controller.connect();   Controller.connect();
```

Figure 3: left: infeasible sequence; right: feasible sequence

target objects and parameters. The sequence on the left is infeasible because no instance of class `Controller` is available (i.e. has been created in advance by preceding calls) for the third method call (`Controller.connect()`). The sequence on the right is feasible as all required target objects and parameter objects are available for each method call.

We formally define sequence feasibility as follows: Let $M = \{m_1, m_2, ..., m_n\}$ be the set of all public methods of the test cluster classes. Additionally, let $r : M \rightarrow C \cup \oslash$ (where C is the test cluster and $\oslash$ is the "no class" element) be a function which assigns each method its return type. Furthermore, let $p : M \rightarrow Pot(C)$, where $Pot(C)$ is the power set of C, be a function which assigns each method the set of required objective parameter types. Finally, let $t : M \rightarrow C \cup \oslash$ be a function which assigns each method its declaring class if it is an instance method, or the $\oslash$ el-

ement, if it is a static method or a constructor. A method call sequence $S =< m_1, m_2, ..., m_j >$ is said to be feasible if

$$\forall i \in 1, 2, ..., j : \forall c \in \{t(m_i)\} \cup p(m_i) \setminus \{\oslash\} \exists k < i : c = r(m_k)$$

otherwise, it is said to be infeasible. In words, a sequence is feasible if preceding method calls, which create the instances that can serve as target object and parameter objects for the method call, exist for each single method call of the sequence.

When designing the representation which is used to encode object-oriented test cases and when defining the corresponding fitness function, call sequence feasibility must be taken into account. The previous approaches in the area of object-oriented evolutionary testing deal with sequence feasibility in different ways: Tonella [11] uses a source-code-like representation and defines six special operators for mutation and crossover working on this representation. One of his mutation operators randomly inserts a new method call into the sequence. Tonella ensures sequence feasibility by inserting recursively additional method calls which create the instances that are required as parameter objects for the method call that the mutation operator inserted. Also, the crossover operator of Tonella does not preserve sequence feasibility per se; hence, additional method calls are randomly inserted. The work of Liu et al. [5] builds upon an Ant Colony Optimization algorithm. Unfortunately, the authors do not describe how they deal with sequence feasibility. In our previous work in this area [12], we used an ID sequence to encode the method calls and mapped the IDs to the actual methods during decoding. Integer mutation and crossover were applied which also produced infeasible sequences since the evolutionary operators were not aware of the call dependences. Therefore, our fitness function used different penalty mechanisms in order to penalize invalid sequences and to guide the search into search space regions that contain valid sequences. Due to the generation of infeasible sequences, the approach lacks efficiency for more complicated cases.

In this paper, we deal with the issue of sequence feasibility by using a representation for method call sequences and evolutionary operators that preserve feasibility throughout the entire search process. We use a tree-based representation which is supported by the majority of today's GP systems and apply already established tree-based mutation and crossover operators. Our fitness function does not need to incorporate a penalty mechanism since no infeasible sequences are generated.

4. STGP

Genetic Programming (GP) is a machine-learning approach to automatically creating computer programs by means of evolution [1]. Given a set of inputs X and outputs Y, a program – or function – p is sought which satisfies $Y = p(X)$. A set of programs is manipulated by applying mutation and crossover unless the optimum program is found or other termination criteria are met.

With a lot of GP approaches, the programs are represented using tree genomes. Figure 4 shows an example tree representing a mathematical expression which uses the input variables $x = (a, b, c)$ where $x \in X$. The leaf nodes of the tree are called *terminals* whereas the non-leaf nodes are called *non-terminals*. Terminals can be inputs to the program, constants or functions with no arguments. Non-terminals are functions taking at least one argument. The *function set* is the set of functions from which the GP system can choose when constructing or manipulating trees. The fitness of a candidate solution is based on its ability to

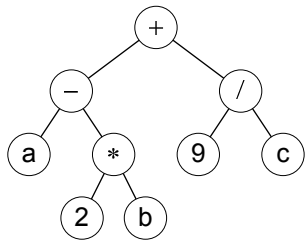

Figure 4: example GP program

satisfy $Y = p(X)$. Let Y_{exp} be the expected known output and Y_p the actual output produced by a program p with $Y_p = p(X)$. Usually, the fitness $f(p)$ of p is calculated using the following formula:

$$f(p) = \sum_{i=1}^{|X|} (p(x_i) - y_{exp_i})^2 \quad (1)$$

where $x \in X$ and $y_{exp} \in Y_{exp}$.

However, GP can also be applied in a non-machine-learning context, e. g. when the set of outputs Y is not known in advance or does not even exist. Then, the fitness calculation must be based on other criteria that express the ability of a program to fulfill a certain task.

Usually, the nodes of a genetic programming tree are not typed. Consequently, the functions of the function set must be able to accept each conceivable argument (the *closure* property of the function set). Since a lot of problems which could be solved by GP are formulated more effectively and efficiently using types, Montana suggests a typing mechanism for GP nodes [8]: each node has a particular return type, i. e. the type of the subtree of which the node is the root. Also, the children are typed, i. e. the arguments of the function the node represents possess particular types. When applying tree construction, mutation, and crossover, the types specify which (non-)terminal can be used as a child of a node and which nodes can be exchanged between two individuals. Montana calls his GP algorithm, that is aware of types, strongly-typed genetic programming (STGP). Strong typing has been extended to support type inheritance and polymorphism (e. g. by the works of Haynes et al. [4], Luke [15], Yu [16]). Consequently, STGP is able to deal with the typing concepts which are inherent to most of the current object-oriented programming languages such as C++ or Java.

5. A NEW APPROACH BASED ON STGP

In this section, we present our approach to automatically generating object-oriented unit test cases. At first, we introduce the *method call dependence graph* as a means to model call dependences. Afterwards, we outline the overall evolutionary algorithm that consists of two optimization levels. How to define the function set and the type set of a GP system is described subsequently. Finally, the last part of this section defines the fitness function we use.

5.1 Modelling Call Dependences

As discussed in section 3, a test program is a sequence of method calls $S =< m_1, m_2, ..., m_n >$. These method calls realize a certain test scenario by creating objects, putting them into particular states, and finally executing the method under test. Each method m_i of the sequence can only be executed if a suitable target object and appropriate parameter objects were created during the method calls m_1 to m_{i-1}. This means that for a single method call, all call dependences for this method – i.e. the preceding creation of the required target and parameter objects – must be satisfied. The call dependences that exist among the test cluster methods can be expressed using a *method call dependence graph* (MCDG). This graph is a bipartite, directed graph whose nodes of type 1 represent methods (method nodes), and the nodes of type 2 represent classes (class nodes). A link between a method node and a class node means that the method can only be called if an instance of the linked class is created in advance. A link between a class and a method means that an instance of the class is created or delivered by the linked method. Figure 5 shows the MCDG for the test cluster of figure 1. For instance, method `Controller.reconfigure(Config)` can only be called if an instance of class `Controller` and an instance of class `Config` are present. An instance of class `Config` can be obtained by either calling `Config(int,int)` or `Controller.getConfig()`. Obviously, the example graph is cyclic. A method call sequence is feasible if it corresponds to an acyclic subgraph of the method call dependence graph. Such acyclic subgraphs can be modeled as trees. Starting at a particular method, the method call dependence graph can be traversed until each branched path has reached a node which has no outgoing link. During traversal, decisions must be made on the class nodes that have multiple out-links: one of the linked method must be selected. An example tree is shown in figure 6. The method node labeled `Controller.reconfigure(Config)`, which is the root node of the tree, is connected to the `Controller` and `Config` class nodes in the MCDG. At the `Controller` class node, there is no decision to make since only `Controller(Config)` delivers an instance of class `Controller`. This method is connected to the `Config` class node meaning that a method must be selected which delivers such an instance. In the tree, the decision was made in favor of `Config(int,int)`. The same is true for the other subpath going from `Controller.reconfigure(Config)` via the `Config` class node to the constructor `Config(int,int)`.

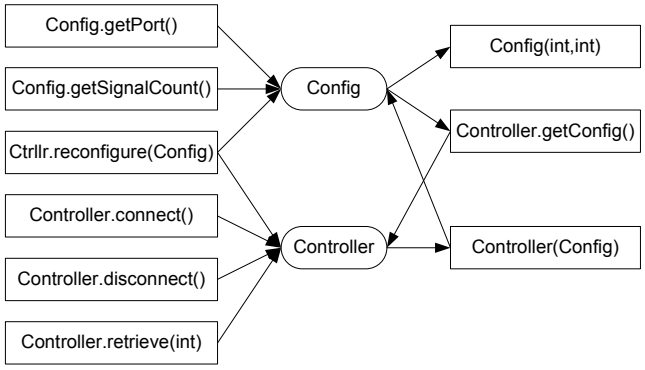

Figure 5: method call dependence graph

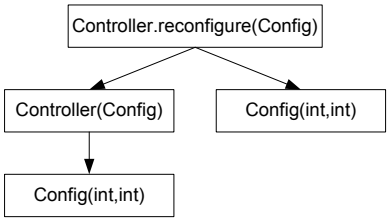

Figure 6: basic method call tree

In the context of strongly-typed genetic programming, it would be easy to configure an STGP system in order to produce trees that are subgraphs of a given method call dependence graph. The trees could be sequentialized and used as method call sequences that are feasible. However, since these trees have only so far considered object construction, there is still the need to integrate "regular" method calls that potentially affect the state of an object but do not necessarily contribute to object construction. An approach for this integration is to view the regular methods as being object-delivering as well: each method can be assumed to take the actual target object as an argument and to deliver the possibly modified object as return value[1]. In terms of the method call dependence graph, additional links can be inserted going from a class node to all method nodes whose target object is of the class node type. Figure 7 shows the extended MCDG for the example test cluster. This allows for a much broader variety of subgraphs and thus method call sequences. Of course, methods which do not return instances and do not affect the state of an object could be eliminated from the graph if they are not currently in the focus of the test. Figure 8 shows a subgraph of the extended MCDG. In the example of figure 8, method `Controller.connect()` (that actually does not return any value) is used to provide an instance used as target object for the call of `Controller.reconfigure(Config)`. This is possible since we assume that the regular methods return their *target object argument* as a return value. If a method already has a return value, this method can appear in a method call tree in two different situations: either an object of the actual return type is required, or an object of the

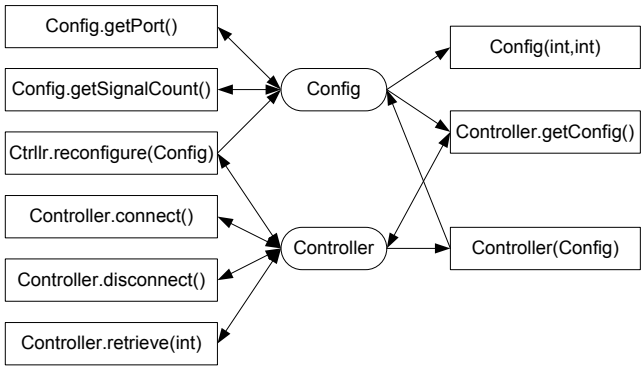

Figure 7: extended method call dependence graph

[1]In some object-oriented languages, such as Modula3, the methods of an object are actually declared with the target object as first argument.

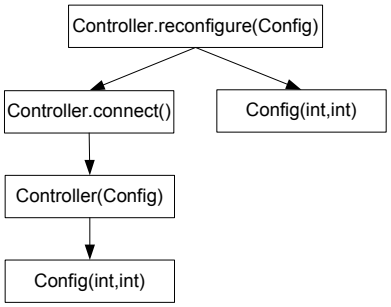

Figure 8: extended method call tree

target object type of this method is required. In both cases, the method fits according to the above assumption.

We are now able to describe the call dependences that exist among the relevant methods using an extended MCDG without losing the ability to also have regular method calls which do not return objects. It is possible to configure a genetic programming system in such a way that it creates and manipulates method call trees which are subgraphs of the extended MCDG of a given test cluster. This configuration consists of the definition of the function set (see 5.3) and the definition of the type set (see 5.4).

A limitation of the above definition of the method call trees is that it does not explicitly support object reuse. This limitation will be dealt with in the following. The method call tree of figure 8 produces two instances of type `Config`; one serves as the argument for the `Controller` constructor, and the other as the argument for method `Controller.reconfigure(Config)`. If we interpret the links between the methods in such a way that the object which is delivered by the link destination method is actually used by the link source method, then it is not possible for one instance to be used as a parameter object for multiple method calls. This case is illustrated in figure 9 (a): the parameter object for the last method call is definitely `cfg2`. However, in

(a)
```
Config cfg1 = Config(int,int);
Controller ctrl = Controller(cfg1);
ctrl.connect();
Config cfg2 = Config(int,int);
ctrl.reconfigure(cfg2);
```

(b)
```
Config cfg1 = Config(int,int);
Controller ctrl = Controller(cfg1);
ctrl.connect();
Config cfg2 = Config(int,int);
ctrl.reconfigure(cfg[1|2]);
```

Figure 9: two ways of interpreting links: (a) strict interpretation; (b) reuse interpretation

practice, there are cases where it is necessary to pass the same instance to multiple method calls in order to cover certain branches of the unit under test. For instance, in case of pointer comparisons, e.g. `if( obj1 == obj2 )`, the condition can only be satisfied if the actual instances represented by the formal parameters `obj1` and `obj2` are identical. Therefore, we assume that a method call tree – although it guarantees that all call dependences are satisfied – leaves the actual relations between the methods open. This *reuse interpretation* leads to an additional degree of freedom when constructing a test program from a method call tree. In figure 9 (b), it is left open which instance is to be used for the call of the last method.

However, at some point the decision must be made which instance should be used. Since for a method call sequence there can be multiple candidate parameter objects for several method calls, many combinations of object assignments are conceivable. Chosing the right combination can be seen as a search problem for which an evolutionary algorithm is employed.

An additional issue to deal with is the optimization of numeric parameter values. Since genetic programming does not primarily focus on parameter optimization, we optimize the numeric parameter values separately from the trees.

In order to deal with the parameter object assignments as well as the numeric parameters, we perform a second level (L2) optimization for each tree individual. We use an evolutionary algorithm that works on vectors of real numbers for the L2 optimization. The following section outlines this two-level optimization approach.

5.2 Test Case Generation Algorithm

As discussed in section 5.1, two optimization levels are necessary: on the first level (L1), method call trees are optimized whereas on the second level (L2), the object assignments and numeric parameter values are optimized. This means that for each tree individual from L1, a complete optimization of its possible object assignments and numeric parameters is carried out. The algorithm for the generation of test cases has the following structure:[2]:

```
begin algorithm generateTestCases
  in: class to be tested CUT
  out: set of test cases T

  identify test cluster TC for CUT
  instrument source codes for TC
  collect test goals TG for CUT
  generate function set for TC
  generate type set for TC

  for each test goal tg in TG
    modify function set for tg
    create initial tree individuals TI
    evaluate tree individuals TI:
    for each tree individual ti in TI
      specify L2 genotype for individual ti
      perform L2 optimization for ti:
        create initial vector individuals VI
        evaluate the vector individuals VI:
        for each vector individual vi in VI
          create a test program out of the tree
           individual and the vector individual
          execute the test program, thereby
           monitor execution flow
          calculate fitness based on distance
        end for
        while termination criterion not met:
          recombine and mutate individuals
          evaluate offspring
```

[2]We do not specify the concrete evolutionary operators here; this is done in the context of our experiments (section 6)

```
        end while
        return fitness of best vi as fitness of ti
    end for
    while termination criterion not met:
      recombine tree individuals
      mutate tree individuals
      evaluate tree individuals
    end while
  end for
end algorithm
```

Initially, the test cluster for the given class under test is defined using static analysis. The concerned classes are then instrumented; thereby, the test goals – in our case all branches of the methods of the class under test – are collected. We follow the goal-oriented approach and carry out a search for a test program for each individual test goal. Before that, the function set and the type set is defined. The function set must be modified for each test goal: it must be ensured that the generated programs contain at least one call of the method that contains the current test goal. For this purpose, the type of the method call tree is defined as the special return type τ of the method under test (see 5.3). During individual initialization, random tree individuals are generated. These individuals are evaluated in the following way: At first, the tree is linearized and all points are computed at which multiple candidate objects are available. For instance, the tree individual shown in figure 8 would be linearized to the sequence shown in figure 9 (b). For each point, a gene is added to the genotype specification of the L2 individual. The number of available candidate objects defines the value ranges for these genes. This happens also for all numeric parameter values. For our example, the following genotype specification S would be created[3]:

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

Then, the L2 optimization is performed in order to search for vectors that correspond to the L2 genotype specification. After creating the initial population, this population is evaluated by creating an executable test program based on the tree individual and the current vector individual, executing this program and calculating the distance of the actual execution flow to the current test goal (the fitness function is dealt with in section 5.5). Vector recombination and mutation take place unless an ideal vector individual is found or another termination criterion applies. The fitness of the fittest vector individual is used as the fitness of the tree individual upon which the L2 optimization is based. Tree recombination and mutation take place unless an ideal individual is found (during the L2 optimization) or another termination criterion applies.

5.3 Function Set Definition

The methods of the test cluster classes constitute the function set that is used by the genetic programming system. The genetic programming system uses the functions from the function set to create (tree-based) individuals. In order to define a function, the name of the function, its return type, and the child types must be specified. We add a method to the function set with the help of the extended MCDG in the following way:

- All method type nodes become functions of the function set.
- The label of a method node is directly used as function name.
- All classes represented by the class nodes which are connected via an out-link from the method node become child types of the function represented by the method node.
- The class represented by the class node that is connected via an in-link to the method node becomes the return type of the function. If there are multiple class nodes connected to the method node via an in-link, the function set entry is duplicated for each class node and the return type is defined appropriately.

The type τ is used as return type of the STGP tree. This type is also used for the function definition of the current method under test. This ensures that this method is called by the method call sequence at least once. An additional function set entry is created for the method under test with this particular return type. Since basic data types are dealt with during the second optimization level, they can be ignored when defining the function set. Table 1 shows the

Function Name	Return Type	Child Types
Controller(Config)	Controller	Config
Controller.reconfigure(Config)	Controller	Controller, Config
Controller.getConfig()	Controller	Controller
Controller.getConfig()	Config	Controller
Controller.connect()	Controller	Controller
Controller.retrieve()	Controller	Controller
Controller.disconnect()	Controller	Controller
Config.Config(int,int)	Config	-
Config.getPort()	Config	Config
Config.getSignalCount()	Config	Config
Controller.reconfigure(Config)	τ	Controller, Config

Table 1: example function set

function set for the test cluster of figure 1. It is assumed that `Controller.reconfigure(Config)` is the method under test that contains the current test goal. The first section of the table shows the function set for class `Controller`. Each method is assumed to return an instance of type `Controller` and to require as a child an instance of type `Controller` except the constructor. Basic data types are ignored. The parameter type `Config` of methods `Controller(Config)` and `Controller.reconfigure(Config)` becomes an additional child type of the corresponding functions. The next table entries are the functions for class `Config`. The last entry is the function representing the method under test. It has the return type τ with which the overall tree is typed.

5.4 Type Set Definition

To account for polymorphic relationships which exist due to inheritance relations, the STGP types used by the function set are specified in correspondance to the type hierarchy of the test cluster classes. We use set-based typing [6, 15] to construct polymorphic types. A set type is a type identifier

[3]The range `int` means the system-dependent range of signed integer numbers.

which is assigned a set of type identifiers. Two types are considered compatible if their type set intersection is non-empty. The type set of an STGP problem consists of the atomic type identifiers and the set types. Figure 10 shows

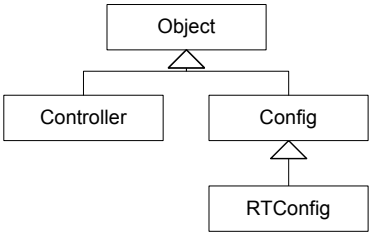

Figure 10: example type set

the class inheritance tree for the test cluster classes and an additional class `RTConfig` which is a subclass of class `Config` (the additional class is introduced for reasons of illustration). Each class of the inheritance tree becomes an atomic type:

$$atomic = \{Object, Controller, Config, RTConfig\}$$

Additionally, a set type is defined for each class that possesses at least one subclass:

$$\begin{aligned} Object\_set &= \{Object, Controller, Config, RTConfig\} \\ Config\_set &= \{Config, RTConfig\} \end{aligned}$$

Whenever a function possesses a child type for which a set type exists, this set type is used instead of the atomic type for child type definition. With the above inheritance tree, the function set of table 1 would be changed in such a way that all `Config` child types would be replaced by `Config_set` child types.

5.5 Test Program Fitness

We use a minimizing (nullifying) distance-based fitness function in order to assess and differentiate the test programs that are generated during the evolutionary search. This function is described in more detail in [13]. The aim of each search is to generate a test program that covers a particular branch of the class under test. The distance of the actual execution flow produced by a given test program in terms of the control flow graph to the target branch can be measured using three different metrics:

- method call distance d_{MC}
- control node distance (approximation level) d_{CN}
- problem node distance (local distance) d_{PN}

The metrics d_{CN} and d_{PN} are the distance metrics that are also used for fitness evaluation in the context of procedural evolutionary testing (e.g. [14]). The metric d_{MC} corresponds to the number of methods that have not been executed due to a runtime exception. In some cases, the randomly generated input values lead to erroneous situations, e.g. when a numeric parameter value is used as an array index and the generated value is negative. Then execution stops at the point where the invalid access occurs. The metric d_{MC} tries to account for these situations. Additionally, in case of an unchecked exception, the exit node of the method in which the exception occurres is considered to be a subtarget of the search and the calculation of d_{AL} and d_{LD} is then based on this subtarget. The fitness f of a test program is a combination of these metrics:

$$f = \lambda d_{MC} + d_{AL} + d_{LD}$$

where λ is a scaling coefficient that is used to weigh the method call distance higher than the other distances. The value of λ must be chosen to be higher than the greatest possible value for d_{AL}. Note that in the case that explicit exceptions are programmatically thrown in the code of the class under test, the throwing statements also become test goals for which test cases are sought.

6. EXPERIMENTS

In this section, we first describe our test environment in short. Afterwards, we present the results of experiments with 4 test objects that demonstrate the efficacy of the approach.

6.1 Test Environment

We implemented our approach with the help of two evolutionary frameworks: we used ECJ (evolutionary computation in Java) provided by Luke [15, 6] for the generation of the method call trees (L1 optimization), and the GEATbx (genetic and evolutionary algorithms toolbox) provided by Pohlheim [3] for parameter object assignment and numeric parameter optimization (L2 optimization). For parsing and instrumenting the Java source code, we employed the OpenJava system provided by Tatsubori et al. [10].

For the experiments, the GP system ECJ was configured as follows: population: 1 subpopulation, 10 individuals; initialization: half/full; selection: tournament; recombination: subtree crossover; mutation: demotion and promotion [2], point mutation; termination: at least after 10 generations. For L2 optimization, GEATbx was configured as follows: population: 4 subpopulations, 10 individuals each; initialization: random values; selection: stochastic universal sampling; recombination: discrete recombination; mutation: real mutation; reinsertion: elitest with generation gap 0.9; termination: if average best fitness over 15 generations does not improve or after 50 generations. While the settings for GEATbx are based on experience with parameter optimization, the settings for ECJ are rather arbitrary.

6.2 Test Objects and Experimental Results

We carried out 4 preliminary experiments in order to validate our approach. Of course, much more experimentation is necessary for a comprehensive validation. Table 2 show the test objects we used and summarizes their complexity characteristics. The last two columns show the results of the experiments averaged over five runs. `Stack` and `BitSet` are taken from the `java.util` package (JDK 1.3), `BoolStack` and `ObjectVector` are taken from package `org.apache.xml.utils` which is part of Xalan 2.7.0. The selection of the classes focused on different complexity issues. Column *LOC* shows the lines of code, column *Max CYC* shows McCabe's cyclomatic complexity of the most complex method, column *#M* shows the number of methods, column *#T* shows the number of targets (branches) which must be covered by test cases, column *Cov.* shows the achieved degree of branch coverage, and column *Exe.* shows the number of test program generated during the search. Full coverage was achieved for all of the test objects. The

Test Object	LOC	CYC	#M	#T	Cov.	Exe.
Stack	142	2	6	10	100%	2671
BitSet	569	11	13	73	100%	5127
BoolStack	200	2	13	16	100%	9393
ObjectVector	429	4	21	52	100%	14330

Table 2: test objects

results are promising and encourage further research and more continuative experimentation.

7. CONCLUSION AND FUTURE WORK

This paper described an approach to automatically generating test cases for structure-oriented unit testing of object-oriented software. Strongly-typed genetic programming was employed for the generation of method call sequences which form the basis of the test cases. Feasibility of the method call sequences is preserved throughout the search process. No fixing of individuals after having applied genetic operators or the use of penalty functions when calculating the fitness is required as it is with previous approaches in this area. We described how to define the function set and the type set of an STGP algorithm: the function set was derived from the signatures of the methods of the test cluster classes; the type set was derived from the inheritance relations of the test cluster classes. We used set-based typing in order to realize polymorphism. Our distance-based fitness function rates the test programs according to their distance to the given test goal. Thereby, it takes runtime exceptions into account. In first experiments, the approach proved successful and produced test cases leading to full branch coverage for four test objects completely automatically.

Further research needs to examine and improve the local distance functions for predicates which are specific to oject-orientation, such as object address comparisons or type checks. Another item for future work is the coverage of test goals which explicitely require a runtime exception to be raised without stopping method call sequence execution. Additionally, it must be investigated how to cover non-public methods by using solely the public interface of a class.

8. REFERENCES

[1] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1998.

[2] 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.

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

[4] T. D. Haynes, D. A. Schoenefeld, and R. L. Wainwright. Type inheritance in strongly typed genetic programming. In *Advances in Genetic Programming 2*, pages 359–376. MIT Press, Cambridge, MA, USA, 1996.

[5] 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.

[6] 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.

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

[8] D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] S. Wappler and J. Wegener. Evolutionary testing of object-oriented software using a hybrid evolutionary algorithm. In *Proceedings of the Congress on Evolutionary Computation (CEC-2006)*, Vancouver, Canada, July 2006. IEEE Press. (to appear).

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

[15] 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.

[16] T. Yu. Polymorphism and genetic programming. In *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038, pages 218–233. Springer-Verlag, 2001.