Evolutionary Testing of Classes

Paolo Tonella
ITC-irst
Centro per la Ricerca Scientifica e Tecnologica
38050 Povo (Trento), Italy
tonella@itc.it

ABSTRACT

Object oriented programming promotes reuse of classes in multiple contexts. Thus, a class is designed and implemented with several usage scenarios in mind, some of which possibly open and generic. Correspondingly, the unit testing of classes cannot make too strict assumptions on the actual method invocation sequences, since these vary from application to application.

In this paper, a genetic algorithm is exploited to automatically produce test cases for the unit testing of classes in a generic usage scenario. Test cases are described by chromosomes, which include information on which objects to create, which methods to invoke and which values to use as inputs. The proposed algorithm mutates them with the aim of maximizing a given coverage measure. The implementation of the algorithm and its application to classes from the Java standard library are described.

Categories and Subject Descriptors

D.2.5 [**Software engineering**]: Testing and Debugging

General Terms

Verification

Keywords

Object-Oriented testing, genetic algorithms, automated test case generation

1. INTRODUCTION

Automated test case generation for Object-Oriented programs is particularly challenging. While a test case for a procedure consists of a sequence of input values, to be passed to the procedure upon execution, and by the expected outputs, test cases for class methods must also account for the state of the object on which method execution is issued. Thus, a test case for a class method includes the creation of an object, optionally the change of its internal state, and finally the invocation of the method being tested, with proper input values. Moreover, the sequence of object constructions and method calls that will be issued by the applications using a given class is in general unknown or only partially known, when a class is being developed. Correspondingly, unit testing of such a class should exercise all relevant alternatives.

Genetic algorithms have been successfully applied to the problem of generating test cases for procedural programs. A population of individuals, representing the test cases, is evolved in order to increase a measure of fitness, accounting for the ability of the test cases to satisfy a coverage criterion of choice. The chromosomes of the individuals being evolved consist of the input values to use in test case execution. When considering the unit testing of classes, a test case must also account for the creation of one or more objects, and the change of their internal state before the final invocation of the method under test. Thus, a more sophisticated definition of the individuals' chromosome is required, and the mutation operators used to evolve a given population must be upgraded accordingly.

In this paper, a representation of the test cases for the evolutionary testing of classes as individuals' chromosomes is proposed. Such a representation includes the specification of a sequence of statements for object creation, state change and method invocation. Thus, a chromosome encodes a sequence of statements, in addition to the input values to be passed as parameters. The classical evolutionary scheme is applied to such chromosomes, with the aim of producing a population that achieves a high level of code coverage. Execution of the resulting test cases is based on the transformation of chromosomes into Junit [14] test methods that can be run automatically.

The proposed method has been implemented in the tool *eToc* and has been successfully applied to some classes taken from the standard Java library. The automatically generated test cases were able to cover code portions that are hard to reach if test cases are to be produced manually. A bug in the standard Java documentation could also be revealed thanks to one of such test cases.

The paper is organized as follows: Section 2 gives the details of the genetic algorithm used for test case generation. Section 3 describes the tool *eToc*, which implements the automatic test case generation method. Experimental results obtained on classes from the Java standard library are presented in Section 4. The next section discusses the related works, while conclusions are drawn in Section 6.

ISSTA'04, July 11–14, 2004, Boston, Massachusetts, USA.

2. GENETIC ALGORITHMS FOR THE UNIT TESTING OF CLASSES

Genetic algorithms have been widely employed to test procedural programs. The adaptations that are necessary to apply them to Object-Oriented code are described in the following. In particular, the focus is on structural testing of a single class.

2.1 Unit testing of classes

During unit testing, each class is considered in isolation, and the possible users of the Class Under Test (CUT) are simulated by means of one or more driver classes. In turn, classes required by the CUT are usually available (bottom-up integration). If this is not the case, they are replaced by stubs.

The procedure typically followed for the unit testing of classes includes the following steps, applied to each method of the CUT and possibly repeated under different execution conditions:

1. An object of the CUT is created using one of the available constructors.
2. A sequence of zero or more methods is invoked on such an object to bring it to a proper state.
3. The method currently under test is invoked.
4. The final state reached by the object being tested is examined to assess the result of the test case.

This procedure can be applied functionally (black-box testing), by deriving the expected final states from the class specifications. Another possibility is to assess the thoroughness of testing by means of a coverage criterion. Traditional coverage criteria (white-box testing) can be used, such as structural (e.g., statement, branch) coverage or data flow (e.g., all-uses) coverage. The steps described above are repeated until a satisfactory level of coverage (e.g., 100% of each method's branches) is reached.

It should be noted that in the first three steps above, parameters may be necessary to invoke either the constructor, some of the methods that change the state of the object under test, or the method under test. If some of these parameters are in turn objects, they must be created and put into a proper state by repeating steps 1 thorough 3 recursively, until all necessary objects are available.

Thus, a test case for the unit testing of a class consists of a sequence of object creations (object under test or parameters), method invocations (to bring objects to a proper state) and final invocation of the method under test. For example, if we are testing method `m` of class `A`, a test case may be:

```
A a = new A();
B b = new B();
b.f(2);
a.m(5, b);
```

where the second object creation is necessary because the second parameter of `m` is an object of class `B` (here and below examples are given in the Java language). Invocation of `f` on `b` aims at changing the state of `b` before passing it to `m`. An assertion on the expected state of `a` after the invocation of `m` should be added to complete the test case.

2.2 Evolutionary testing

In this section the algorithm for evolutionary testing originally presented in [16] is summarized (with slight modifications). The changes and adaptations in chromosomes, mutation operators and input generators necessary to use it for unit testing of classes are described in the next sections.

```
testCaseGeneration(classUnderTest: Class)
1   targetsToCover ← targets(classUnderTest)
2   curPopulation ← generateRandomPopulation(popSize)
3   while targetsToCover ≠ ∅ and
                executionTime() < maxExecutionTime
4       t ← selectTarget(targetsToCover), attempts ← 0
5       while not covered(t) and attempts < maxAttempts
6           execute test cases in curPopulation
7           update targetsToCover
8           if covered(t) break
9           compute fitness[t] for test cases in curPopulation
10          extract newPopulation from curPopulation
                according to fitness[t]
11          mutate newPopulation
12          curPopulation ← newPopulation
13          attempts ← attempts + 1
14      end while
15  end while
```

Figure 1: Genetic algorithm for test input generation based on levels of coverage.

Figure 1 contains the macro steps of the algorithm. Input data are generated so as to satisfy a given coverage criterion for the methods in the CUT. The set of targets (e.g., branches) to be covered is determined at line 1. Then, a set of test cases is randomly generated (*curPopulation*). The exact form of a test case (chromosome) will be clarified in the next section. For the time being, the intuitive description given above suffices: a test case is a sequence of object constructions and method invocations, each complete with parameter values.

The algorithm generates new test cases as long as there are targets to be covered or a maximum execution time is reached (line 3). For each target to be covered, selected at line 4, a maximum number (*maxAttempts*) of successive generations are produced out of the initial population (loop at line 5). Test cases in the current population are executed at line 6, possibly covering some of the previously uncovered targets. The targets still to be covered are updated at line 7, and if the currently selected target (t) has been covered, the most internal loop is exited and a new target is selected. If the current target was not covered by any execution of the test cases in the current population, a measure of proximity of each test case to the target is computed (*fitness*[t], line 9). A new population is randomly extracted with replacement from the current population. Each test case has a probability of being selected which is proportional to its fitness with respect to the current target t. In other words, test cases that come closer to the target are more likely to be extracted in the formation of the new population. Then, the new population is mutated according to proper mutation operators (described in detail in a next section), and another attempt to cover the current target is made (if *maxAttempts* has not yet been reached).

During the execution of the algorithm in Figure 1, each time a previously uncovered target is covered by a test case

in the current population, the test case is saved as one of those necessary to achieve the final level of coverage. The output of the algorithm is thus the set of test cases that allowed covering at least one new target. Such a set may be redundant, in that test cases that are inserted later in the resulting set may cover targets that were previously associated to a different test case. Thus, if all the targets covered by a given test case are also covered by test cases added later to the resulting set, the former does not contribute to the final level of coverage any longer and can be removed from the final set. In order to cope with such a situation, a post-processing of the set of resulting test cases is performed, aimed at minimizing it. The minimization procedure is a simple greedy algorithm, which iteratively selects the test case that gives the largest increase in the number of covered targets. Such a test case is added to a minimized set of test cases (initially empty) until the final coverage level is reached. All test cases not added during the post-processing are redundant and can be dropped.

The two critical choices in the algorithm in Figure 1 are the measure of fitness of each test case and the mutation operators. These are the two factors that drive the evolution of the current population into a new population that has more chances to cover the current target. The fitness determines the probability of an individual (test case) to survive and participate, in an evolved form, in the new population, while the mutation operators determine how new individuals (test cases) are generated out of the existing ones.

Mutation operators that are appropriate for the evolutionary testing of classes are discussed in detail below. As regards the fitness measure, several choices are possible. Following the indications in [16] and assuming that the adopted coverage criterion is structural (e.g., branch coverage), in this work the fitness of a test case with respect to a given target is obtained from the control and call dependence edges that are traversed during its execution. Specifically, given the transitive set of all control and call dependences that lead to the given target, the proportion of such edges that is exercised during the execution of a test case measures its fitness. Thus, the fitness will be close to 1 for the test cases which traverse most of the control and call dependence edges that lead to the target, while it will be close to zero when the execution follows a path that does not intersect with edges leading to the target.

The parameters of the algorithm in Figure 1 that can be fine tuned externally for each CUT are the maximum execution time (*maxExecutionTime*) and the maximum number of attempts per target (*maxAttempts*), as well as the population size (*popSize*). They can be augmented when the achieved coverage level is not satisfactory.

2.3 Chromosomes

The structure of the individuals' chromosomes is quite simple when evolutionary testing is applied to procedural programs. In fact, it usually consists of the sequence of input values to be provided to a program during its execution. On the contrary, a test case to be used for the unit testing of a class is not just a sequence of input values. Typically, the *main* method does not exist at all, when a class is considered in isolation. The class must be tested with reference to its interface constructors and methods, and with no other knowledge about the sequence of method invocations that will be actually issued by the final application(s) using it. Thus, a test case is a sequence of constructor and method invocations, including parameter values. Assertions on the expected state after their execution completes the test case. The chromosomes to be used for evolutionary testing of classes should thus include both the specification of the sequence of operations to be performed and of the associated parameter values. Such a sequence is not fixed and must be constructed according to the same evolutionary rules that characterize the selection of input values.

Figure 2 defines the syntax of chromosomes. Curly braces and question mark are meta-characters used for optional expansions, while non terminals are in angular brackets.

```
<chromosome>  ::=  <actions> @ <values>
<actions>     ::=  <action> {: <actions>}?
<action>      ::=  $id = constructor ( {<parameters>}?  )
                |  $id = class # null
                |  $id . method ( {<parameters>}?  )
<parameters>  ::=  <parameter> {, <parameters>}?
<parameter>   ::=  builtin-type {<generator>}?
                |  $id
<generator>   ::=  [ low ; up ]
                |  [ genClass ]
<values>      ::=  <value> {, <values>}?
<value>       ::=  integer
                |  real
                |  boolean
                |  string
```

Figure 2: Syntax of chromosomes.

A chromosome is split into two parts, separated by the character `'@'`. The first part (nonterminal <*actions*>) contains a sequence of constructor and method invocations, separated by the character `':'`. The second part contains the actual input values (comma-separated) to use in such invocations, and corresponds to the chromosomes traditionally used in evolutionary testing of procedural programs.

Each <*action*> can either build a new object, assigned to a chromosome variable (indicated as `$id`), or it can issue a call to a method on an object identified as `$id`. A special case of object assignment involves the value `null`, corresponding to a reference that points to no object (no object allocation occurs). The second production associated with <*action*> accounts for this case. The `null` value is preceded by the class of `$id` (separated by the character `'#'`), because this variable can be used as a parameter only when its type matches that of the formal parameter (`null` has no type information associated).

Parameters of method calls (see <*parameter*>) or constructor invocations can be built-in types, such as `int`, `double`, `boolean`, or chromosome variables (`$id`). Class `String` is considered a built-in type, in that values are generated for it without resorting to an explicit constructor invocation. The generation method employed to produce the value of a given parameter can be optionally indicated in square brackets (<*generator*>). A number can be randomly chosen in an interval ranging from **low** to **up** or (second production) an external class produces the input value, in accordance to arbitrarily complex value generation rules. In absence of a generator specification, the default generator is used. Input generators are discussed in more detail in a next section.

The second portion of a chromosome contains input values, which belong to a built-in type, and thus have the form of an integer, a real number, a boolean or a constant string.

Not all chromosomes that can be constructed in accordance with the rules in Figure 2 are well-formed. A chromosome is well formed if:

1. Chromosome variables (indicated as `$id`) are always assigned before being used (as parameters or targets of method calls).
2. For each built-in type in the actions, an input value of associated type occurs in the second half of the chromosome, at corresponding position.

Moreover, a chromosome represents an actual execution sequence for a CUT if all involved methods and constructors exist and they have a signature that matches the types specified in the chromosome. For example, the chromosome:

```
$a=A():$b=B():$a.m(int, $b) @ 3
```

is well formed and represents an actual execution sequence if classes `A` and `B` have a constructor with no parameters, and if class `A` defines a method `m` whose first parameter is of `int` type and whose second parameter is an object of class `B` (or any supertype). A variant of this chromosome may consider changing the state of the object `$b` before passing it to `m`. Assuming that a method `f` exists in class `B`, the state change of `$b` can be obtained by executing the following chromosome:

```
$a=A():$b=B():$b.f(String):$a.m(int, $b) @ "asd", 3
```

Invocation of `f` requires that a `String` parameter is generated and passed to it. The second part of the chromosome specifies it (`"asd"`).

If method `m` handles also a null value as second parameter, the following alternative chromosome can also be used to test class `A`:

```
$a=A():$b=B#null:$a.m(int, $b) @ 3
```

In the examples above, values `3` and `"asd"` have been generated by the default integer and string generators. If two ad-hoc generators are used instead, chromosomes will have the following form:

```
$a=A():$b=B():$b.f(String[DateGenerator])
:$a.m(int[-2;2], $b) @ "3/3/2003", -2
```

The parameter of `f` is produced by class `DateGenerator`, which generates strings that represent dates, while the first parameter of `m` is randomly selected in the range from -2 to 2 (inclusive).

2.4 Mutation operators

Chromosomes are transformed by the genetic algorithm in Figure 1 by means of randomly chosen mutation operators. In the following, a set of mutation operators that can be used for the evolutionary testing of classes is described.

Mutation of input value:
A value is replaced by a randomly generated value of the same type.

Example:
`$a=A():$b=B(int):$b.f():$a.m(int,$b)@ 1 ,5`
`$a=A():$b=B(int):$b.f():$a.m(int,$b)@ 6 ,5`

The `int` value passed to the constructor of class `B` is changed from 1 to 6.

Constructor change:
One of the constructors in an action is randomly changed. Previously required values or objects (not used elsewhere) are dropped. New required values or objects are inserted. Existing objects are possibly reused.

Example:
`$a=A():$b=B(int):$b.f():$a.m(int,$b)@ 1 ,5`
`$a=A():$c=C():$b=B($c):$b.f():$a.m(int,$b)@5`

The constructor in the second action is replaced by another constructor from the same class (`B`). The old constructor has an input parameter of type `int` and value 1 (first value in second half of chromosome), which is removed when the constructor is changed. The new constructor has a parameter of class `C`, thus it requires a new chromosome variable (`$c`) holding the object produced by the invocation of a constructor of class `C`.

Insertion of method invocation:
New method invocations are inserted. Values or objects necessary as invocation parameters are also inserted.

Example:
`$a=A():$b=B(int):$b.f():$a.m(int,$b)@1,5`
`$a=A():$b=B(int):$b.g(int):$b.f():$a.m(int,$b)@1, 4 ,5`

A chromosome variable pointing to an object is randomly chosen (in our example, `$b`), and a method call is inserted between its assignment and its last use. The insertion point in the example above is before the call of method `f` on `$b`. Since the new call (method `g` of class `B`) requires an `int` parameter, a new integer value is inserted in the second half of the chromosome, in the position corresponding to the parameter of the new call.

This mutation operator is applied repeatedly, so as to insert a number of invocations possibly greater than or equal to 1. The probability of an insertion decades exponentially with the number of insertions n performed so far, according to the function: 0.5^n. This means that after each insertion a random decision is made about the re-application of this mutation operator.

Removal of method invocation:
Some method invocations (excluding the last one) are randomly selected and removed. Previously required values or objects are dropped.

Example:
`$a=A():$b=B(int):$c=C(int):$b.h($c):$b.f():$a.m(int,$b)`
`@1, 4 ,5`
`$a=A():$b=B(int):$b.c2():$a.m(int,$b)@1,5`

Invocation of method `h` on object `$b` is randomly selected for removal. Since this is the only use of the chromosome variable `$c`, this variable is dropped. Its constructor has an `int` parameter, whose value is also removed from the second half of the chromosome.

Similarly to the insertion of method invocations, this mu-

tation operator is also applied repeatedly. The probability of removing another method invocation after n previous removals is 0.5^n.

One-point crossover:
After joining two chromosomes cut at a randomly selected middle point (after the constructor of the target object and before the last method invocation), unnecessary constructors/method calls are removed and missing constructors are added. Conflicting variables of different types are renamed.

Differently from the other mutation operators, working on single chromosomes, the one-point crossover transforms two chromosomes into two mutated chromosomes.

Example:
`$a=A():$b=B():$b.f():$a.m(int,$b) @1,5`
`$a=A(int,int):$b=B():$b.g():$a.m(int,$b) @0,3,4`

`$a=A():$b=B():$b.f():$b.g():$a.m(int,$b)@1,4`
`$a=A(int,int):$b=B():$a.m(int,$b) @0,3,5`

The two chromosomes at the top are transformed into the two chromosomes at the bottom by means of the crossover operator. The cut point in the first chromosome is immediately after the call of `f`, while the cut point in the second chromosome is immediately before the call of `g`. The tails of the two cut chromosomes are swapped, producing the results at the bottom. Input values are cut and swapped correspondingly.

In the example above, all chromosome variables used at some point are properly defined, no unused variable exists, and there is no conflict between variables, so that it is not necessary to adjust the two resulting chromosomes.

An example which requires the insertion and the removal of a constructor invocation is the following (swapped chromosome tails are underlined):

`$a=A():$b=B(int):[$c=C(int)]:$b.h($c):$b.f():$a.m(int,$b)`
`@1,[4],5`
`$a=A(int,int):$b=B():$a.m(int,$b) @0,3,6`

`$a=A():$b=B(int): $a.m(int,$b)@1,6`
`$a=A(int,int):$b=B():[$c=C()]:$b.h($c):$b.f():$a.m(int,$b)`
`@0,3,5`

Since the first chromosome after crossover does not use `$c` any longer, the action `$c=C(int)` is removed together with the associated value, 4. The second chromosome produced by crossover uses a variable `$c` of class `C` which is undefined. Thus, before its first use an invocation to a constructor of class `C` is inserted.

An alternative form of crossover would consist of retrieving the constructors for the undefined variables used in the tails from the original chromosome heads, instead of inserting brand new constructors for them.

2.5 Input generators

The input generators to use in chromosome construction are specified with the declaration of the method signatures. If only type names are indicated in the signatures, the default input generators are used. Customization is possible by specifying (in square brackets) parameters for the default generators or a custom generator class. The following method declarations involve only default generators:

```
A.A(int,int)              B.f(boolean)
B.B(String)               A.m(int,B)
```

The rules for input generation used by the default generators are the following:

Integer and real numbers: *Integer and real numbers are uniformly chosen in the interval* $[0, 100]$.

Boolans: *Boolean values* `true` *and* `false` *are randomly chosen with equal probability (0.5).*

Strings: *Characters inserted into the generated string are uniformly chosen among the alphanumeric characters ([a-zA-Z0-9]). The probability of inserting another character after inserting n characters is* 0.5^{n+1}.

Thus, the empty string ($n = 0$) is generated with probability 0.5. At least one character (uniformly chosen) is inserted with probability 0.5. Given a string of length 1, a second character is inserted with probability 0.25, while the given string is complete with probability 0.75, and so on.

In the declaration of method signatures, built-in types are optionally followed by parameters (in square brackets) used by the default input generators. For example, the following constructor: `B.B(int[-2;2])` instructs the generator for the integer type to uniformly choose a value in the interval [-2, 2] instead of [0, 100].

In order to define a completely customized input generator, the name of a class can be specified, as in: `B.B(int[MyIntGenerator])`, where class `MyIntGenerator` must implement the interface `IntGenerator`, requiring the definition of the body of method `newIntValue()`. Similarly, custom generators can be defined for real numbers, booleans and strings, by implementing the respective interfaces.

When the `null` value is acceptable as a parameter, the related generator can be instructed to produce `null` values with a given probability. For example, the following method signature: `A.m(int,B[null;10%])` specifies that the second parameter of `m` can be `null`, and that `null` values are expected to be generated with probability 0.1.

The possibility to customize the input generators is extremely important in cases where the methods in a class work properly only if parameter values satisfy certain constraints. Class generators provide the maximum flexibility to realize them.

3. THE TOOL *ETOC*

The genetic algorithm for test case generation has been implemented in the tool *eToc* (evolutionary Testing of classes) for the Java language. The high level organization of the tool is depicted in Figure 3.

The Java source code of the CUT is analyzed and transformed by the *Branch instrumentor* module, which produces an instrumented version of the original class and determines information about method signatures, call and control dependences, used by other modules. In particular, the *Chromosome former* is able to generate chromosomes based on the method signatures, and implements the mutation operators to transform them.

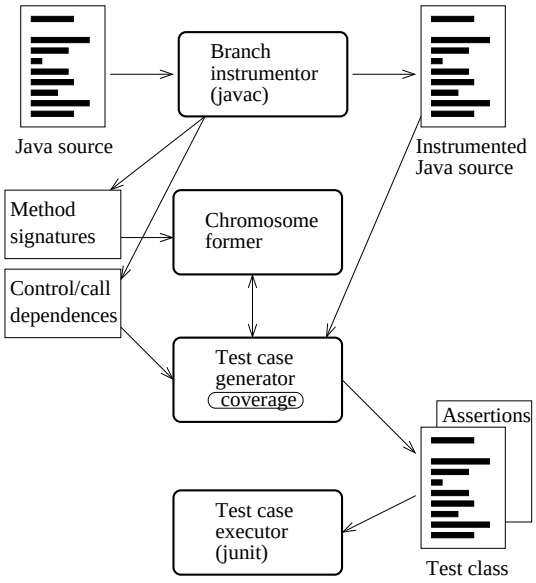

Figure 3: High level architecture of *eToc*.

The *Test case generator* evolves a population of chromosomes according to the rules of the genetic algorithm described in Section 2. Chromosomes are obtained from the *Chromosome former*, which is also responsible for mutating them. The test case associated to each chromosome is executed using the instrumented version of the CUT, in order to determine the targets (branches) covered by each individual.

At the end of execution, the *Test case generator* minimizes the set of chromosomes while keeping the level of coverage reached by the genetic algorithm. Minimized test cases are converted into the format used by the tool Junit [14], that is, a Junit test class is generated. After manually adding assertions, test cases in the test class can be executed by means of Junit.

3.1 Branch instrumentor

In the instrumented code produced by the *Branch instrumentor*, every control flow branch is uniquely identified and traced during execution, under the assumption that the adopted coverage criterion be branch coverage.

The *Branch instrumentor* has been implemented as a visitor of the Abstract Syntax Tree (AST) produced by the Java compiler *javac*. Specifically, the pretty printer, that is distributed with *javac* and is implemented as an AST visitor, has been customized to print the instrumented code.

3.2 Chromosome former

The *Chromosome former* constructs new chromosomes and mutates existing chromosomes based on information about method signatures. When requested to produce a new chromosome, the method or constructor that contains the current target is specified, so that its invocation be the last action encoded in the chromosome.

All mutation operators described in Section 2 are implemented by the *Chromosome former*. Moreover, all default input generators are available, including their parameterized versions. Custom input generators are used when the name of a generator class is specified in square brackets. In such a case, the *Chromosome former* exploits the reflection facilities of Java to call the generation method from the generator class (e.g., `newIntValue()` for an `IntGenerator`). Null values are also generated if specified in the signature description, according to the indicated probability.

When the signature of a method includes interfaces (e.g., `Comparable`) or abstract classes (e.g., `AbstractList`), the *Chromosome former* can be instructed on which concrete class to use in order to construct an object of the interface type.

3.3 Test case generator

Execution of the test case associated to each chromosome is performed by the *Test case generator* within the inner loop of the genetic algorithm in Figure 1. The actions encoded in a chromosome are converted into constructor and method invocations, that can be actually issued thanks to the reflection capabilities of Java. The instrumented version of the CUT is used in such executions, so that after each run, the execution trace can be accessed. The overlap between execution traces and control/call dependences leading to the current target determines the fitness of the individuals.

After minimization, the resuting set of chromosomes is converted into the format prescribed by Junit [14]. The resulting test class is manually edited, in order to add assertions inside the body of each test method. Knowledge about the CUT's requirements/specifications is necessary to augment the test cases with assertions. This is the most human-intensive phase in the whole testing process. It happens only at the end of the test case generation process, when the algorithm stops and returns the test suite.

Below is an example of test method (`testCase2`) produced by the *Test case generator* for a binary tree class, with a manually inserted assertion. It was obtained from the following chromosome:

```
$x0=BinaryTree():$x1=Integer(int):
$x2=BinaryTreeNode($x1):$x0.insert($x2):
$x3=Integer(int):$x0.search($x3)@5,5

public void testCase2() {
  BinaryTree x0 = new BinaryTree();
  Integer x1 = new Integer(5);
  BinaryTreeNode x2 = new BinaryTreeNode(x1);
  x0.insert(x2);
  Integer x3 = new Integer(5);
  assertTrue(x0.search(x3));
}
```

A `BinaryTree` is first created. Then a `BinaryTreeNode` is inserted into the tree, pointing to an `Integer` datum holding the value 5. The final method invocation is a `search` of an `Integer` object holding the value 5, which is expected to succeed (return value `true`).

3.4 Test case executor

Once test cases are encoded as methods of a Junit test class, their execution is directly provided by the tool Junit, which produces summary information about passed and failed test cases, as well as about the specific assertion(s) that are not satisfied. The test suite produced by the *Test case generator* and run by Junit is guaranteed to provide the level of coverage reached by the genetic algorithm.

4. EXPERIMENTAL RESULTS

Experimental results on the proposed method have been obtained by employing the tool *eToc* for the unit testing of some of the classes belonging to the standard Java library.

4.1 Procedure

First of all, CUTs were instrumented by means of the module *Branch instrumentor* of *eToc*. The output file with method signatures was refined, by manually inserting the names of the classes that actually implement any interface in the signatures.

Then, the *Test case generator* produced a set of Junit test cases for each CUT, maximizing the level of branch coverage. Assertions were inserted manually into the test methods, according to the folllowing procedure:

- Each time a method call returns a value, an assertion is added, comparing actual and expected value.
- Each time a method call might possibly throw an exception, a `try-catch` block is added, surrounding it. A flag `thrownException`, initialized to *false*, is set to *true* inside the `catch` block, so that an assertion can be added to check if the exception was actually raised.
- At the end of each test case, the final state of the object under test is examined. For example, if the CUT implements a method `toString` that returns a `String` representation of the object, it can be invoked to get (part of) the final object state, to be contrasted against the expected state in a final assertion.

The capability of the resulting test suite to spot the presence of defects was evaluated by means of the *fault seeding* method. Artificial defects have been manually inserted into the code of the CUTs, and the related test suites have been re-executed by means of Junit. The occurrence of failures/errors indicates that test cases were able to reveal the defects.

4.2 Classes under test

Class	LOC	pub. meth.
StringTokenizer	313	6
BitSet	1046	26
HashMap	982	13
LinkedList	704	23
Stack	118	5
TreeSet	482	15

Table 1: Features of the classes under test, all taken from `java.util`, Java 2 Std. Ed. version 1.4.0.

The classes used to evaluate the proposed technique were taken randomly from the standard Java library distributed with the Software Development Kit (SDK) version 1.4.0. Their features are summarized in Table 1. The size of these classes covers the typical range, going from around 100 Lines Of Code (LOC) to around 1000 LOC (in the package `java.util` the class size is between 15 LOC and 2429 LOC, with an average of 559 LOC). The number of methods varies accordingly. The internal complexity of the CUTs is also quite variable, and is well reflected in the LOC.

The smallest class, `Stack`, is quite simple. It implements a stack by extending a base class, `Vector`, and adding a few short methods with a quite simple internal structure. At the other extreme, the class `BitSet` is quite complex. It handles sets represented as sequences of bits. Dynamic allocation of words of bits is implemented in this class. The internal logics of its methods is quite complicated. Set capacity has to be increased when required by the ongoing operation. Moreover, multiple words for each bit set have to be managed seamlessly. All of this makes the code for processing the operations on bit sets quite complex.

The other classes have intermediate characteristics. `TreeSet` implements sorted sets based on a tree representation of objects provided by another associated class, `TreeMap`. Class `StringTokenizer` implements string manipulation operations aimed at breaking strings into tokens.

Classes `HashMap` and `LinkedList` implement the two classical data structures of hash tables and doubly linked lists. `HashMap` requires careful management of the hash table capacity and load factor. Moreover, insertion and extraction of `null` objects need additional logics in the implementation of some methods. Methods in class `LinkedList` manipulate the references to next and previous list elements.

Class `HashMap` has been tested in two different contexts, associated to two different kinds of objects used as keys of the hash table. In the first context, objects of type `Integer` are used as hash table keys, while in the second context `String` objects are used. In the following, the two different test settings for `HashMap` are indicated as `HashMap1` and `HashMap2`.

4.3 Results

The main parameters of the genetic algorithm in Figure 1 (*maxAttempts, maxExecutionTime, popSize*) have the following default values: (10, 60s, 10). When the generated test cases do not reach a satisfactory level of coverage using default values, these are increased to (20, 600s, 10). This happened with classes `LinkedList` and `HashMap`. When results are not yet satisfactory, parameters are increased to (30, 3600s, 20). This occurred only in one case (class `BitSet`).

As apparent from these values, the upper limit for the execution time is adequate to the testing phase. A test generation activity that requires between a few minutes and one hour is typically acceptable. Times have been obtained on a Pentium PC with a 3 GHz processor and 1 Gb of central memory, under normal load conditions. Initial test cases are generated randomly, by inserting a constructor for the object under test and an invocation of the method containing the current target. Required object parameters are obtained by inserting proper constructor invocations into the chromosome.

Table 2 shows some data about the test case generation

Class under test	Coverage	Test cases	Execs	Time (s)
StringTokenizer	11/11	3	820	2
BitSet	172/177	28	38700	4930
HashMap1	39/41	8	4380	1338
HashMap2	39/41	7	4310	697
LinkedList	56/57	9	25690	2034
Stack	10/10	2	230	1
TreeSet	20/21	4	2480	60

Table 2: Results of test case generation.

process. The level of coverage refers to the branches in the public methods (accounting for around 2/3 of the code of all methods). It measures the number of covered branches over the total number of branches. The number of test cases accounts for the minimization algorithm that eliminates redundant test cases. The actual execution time (last column of Table 2) exceeds, in some cases, the time limit reported above. This occurs because the check on the time expired so far is done only after completing the attempts to cover a target, up to the maximum number of attempts allowed (see Figure 1).

The number of generated test cases ranges from 2 to 28 (for class `BitSet`), being below 9 in all but one case. This indicates that the test suites produced by the proposed method are quite compact. Augmenting them with assertions is thus expected to require a minor effort. The only exception is `BitSet`, for which 28 test cases were produced. The reason for this relatively high number is the complex computational structure of the methods in this class. Covering all its branches is a challenging task, and a programmer is expected to spend a lot of time to find inputs satisfying all conditions that are handled in alternative execution branches. They are related to the allocation of extra bit words for the result of some operations on bit sets, or, symmetrically, their deallocation. The related logics is very complex.

Branches that were not covered by any of the generated test cases have been examined in more detail. Their total number is 11. Five of them are infeasible, and correspond to a code fragment that catches the exception: `CloneNotSupportedException`, that may be raised when method `clone` is called from the superclass of current class (`super.clone()`). Since all 5 classes with this uncovered branch inherit method `clone` from class `Object`, where no such exception is thrown, the branch is infeasible.

One uncovered branch in class `HashMap` (both in `HashMap1` and `HashMap2`) can be actually executed only when more than 1 billion items are inserted into a hash table. No test case generated according to the proposed method can ever produce such a test case. The remaining 4 uncovered branches belong to `BitSet`, and are also associated to test cases that cannot be generated in the described setting, since they require that a method is called with a parameter equal to the `this` object or with a parameter that does not belong to class `BitSet`.

From the discussion above it is clear that the coverage level reached by the generated test cases is "practically" equal to 100%, due to the presence of infeasible branches and of branches that can be covered only in a different setting. Thus, the capability of the proposed genetic algorithm to produce test cases for branch coverage of a class is confirmed by the experimental data.

Assertions have then been added to the test cases, according to the procedure described above. For example, the final statement of one of the test cases for class `LinkedList` was changed into: `assertTrue(x0.remove(x3))`. In fact, `x3` is assigned `null` in a previous statement of this test case, and `null` is inserted into the `LinkedList` `x0` by another previous statement. Thus, the call to `remove` is expected to return `true`, since the removed element is contained in the list. The following assertion about the final state of `x0` was added at the end of the test case:

```
assertTrue(x0.toString().equals("[2]"));
```

On average, 7 assertions have been added manually for each test case, following the procedure described at the beginning of this section, with the time necessary to insert them the order of a few minutes. Further automation can be achieved also in this phase by generating a skeleton of the assertions to be inserted. For example, the try-catch block surrounding method calls that can raise exceptions could be produced automatically.

Execution of the test cases augmented with assertions revealed an interesting surprise. A bug in the documentation of one of the CUTs was identified. The description of the behavior of method `addAll(int index, Collection c)` from class `LinkedList`, available in the Javadoc documentation, ends with the following sentence:

> *Throws: IndexOutOfBoundsException – if the specified index is out of range (*`index < 0 || index > size()`*).*

Correspondingly, the call `x0.addAll(64, x1)` was embedded into an assertion checking that an exception be actually thrown (the size of `x0` is less than 64). The assertion failed, the reason being that no exception is raised if the parameter `c` of the call is an empty `Collection`. Such a behavior is perfectly acceptable, thus the bug was regarded as a documentation defect, rather than a code defect. The corrected documentation of method `addAll(int index, Collection c)` should be as follows:

> *Throws: IndexOutOfBoundsException – if the specified index is out of range (*`index < 0 || index > size()`*) and* `c` *is not empty.*

Class under test	Failures or errors
StringTokenizer	4/5
BitSet	3/5
HashMap1	4/5
HashMap2	3/5
LinkedList	4/5
Stack	5/5
TreeSet	4/5

Table 3: Results of fault seeding.

Five faults have been manually seeded into each CUT, trying to simulate typical programming errors (part of a condition missing, wrong relational operator, etc.). The proportion of them which resulted in an assertion failure or an execution error during test case execution with Junit varied from 3/5 (2 cases) and 4/5 (4 cases), to 5/5 (1 case). Table 3 reports the data obtained on each of the considered CUTs.

The capability of the automatically generated test suites to spot defects is thus moderately high, although not optimal. Faults that remain uncovered are mainly associated to data flows associated to some attribute of an object or to its state changes. Coverage of all method branches seem not to imply the ability to catch such cases. However, definitive results on this issue can be obtained only by performing a more extensive fault seeding experiment.

5. RELATED WORKS

Test data generation based on the minimization of a cost function associated with each branch predicate was investigated in [7, 8, 12, 13]. In [9, 10] inputs satisfying the branch conditions are obtained by solving a set of associated linear constraints and using the results to refine previous values. Genetic algorithms have been used to achieve the minimization of cost functions associated with branches in [1, 15, 16, 17]. Usage of a moving target to satisfy coverage criteria was proposed in [16], while proper fitness measures to estimate the distance from a given execution branch are studied in [1]. All these works assume a procedural programming style.

Symbolic execution and automated deduction are used in [5] for the testing of classes considered in isolation. The aim of this work is exercising the def-use pairs associated with the instance variables of a class. Symbolic execution is exploited to obtain the method pre-conditions that must be satisfied in order to traverse a feasible, def-clear path for each def-use pair. Automated deduction is used to determine the ordering of method invocations that allows satisfying the preconditions of interest.

In [3] a genetic algorithm (improved in [2]) is employed to optimize an initial test suite, in the context of mutation testing. The ability of the test cases to *kill* mutants (i.e., faulty variants) of the original program is increased by evolving them according to the genetic algorithm. Then, the final test suite is executed on each mutant, with pre-conditions, invariants and post-conditions attached to methods (design-by-contract). The cases where no pre/post-condition or invariant is violated by a mutant hint for possible (manual) improvements of the contract, which should be made more specific.

The main differences between the present work and [3] descend from the different application context (mutation testing vs. coverage testing). In [3] an individual which is subject to evolution is a whole test suite (instead of a single test case). This changes completely the meaning of the crossover operator, as well as the aim of the selection process. Moreover, the mutation operators considered in [3] are a subset of those proposed here, input generators are not discussed at all and simplifying assumptions are made on method parameters (the general case where a parameter is itself an object is not considered).

The approach described in [4] is based on the availability of class invariants and method pre/postconditions. The input space, properly finitized, is explored to determine all its points that satisfy the invariants/preconditions. In order to make such an exploration efficient, the access of the invariant/precondition predicate to class fields is monitored to prune large portions of the search space and only non-isomorphic inputs are generated. Then, postconditions are used as execution oracles. While the generation procedure used in [4] builds complex data structure directly, by satisfying the related class invariants, in this paper only elementary data types are built through generators, while complex data structures are obtained from the execution of (part of) the chromosome, by means of constructor and method invocations encoded inside it.

In this paper, the focus is on the adequacy of the test cases with respect to a coverage criterion of choice. The problem of the oracle, which is orthogonal to the problem of test case generation, is handled by manually adding assertions. Work on the automated recovery of likely invariants or algebraic specifications by means of dynamic analysis [6, 11] might be a good complement, tackling such a problem. In [11], dynamic analysis is based on the execution of terms (similar to the chromosomes used in this paper) that are grown incrementally, beginning with a constructor and adding method invocations. An evolutionary approach, such as the one presented here, could be beneficially integrated into this term construction procedure.

6. CONCLUSIONS AND FUTURE WORK

Usage of a genetic algorithm for the unit testing of classes was extremely powerful. Optimal coverage of the public method branches was achieved within a reasonable computation time, and the resulting test suites were generally compact. The sequences of object allocations and method invocations, with the respective input values, that are inserted into the generated test cases exercise the CUTs in a sophisticated way. In order to achieve branch coverage, they explore all sides of the boundary conditions in alternative execution flows. Some of these conditions are very hard to satisfy, and a programmer expected to produce a test case for them would have a hard job. This is especially true for class `BitSet`, where the logics in the allocation and deallocation of multiple words associated to bit sets is quite complex.

Future work will be devoted to investigating alternative coverage criteria, such as the data flow criteria, which are expected to improve the fault exposing capability of the automatically generated test cases. The possibility to use genetic algorithms for state based testing of classes will be also studied. The extension of the proposed methodology for multi-class testing is another interesting topic of future research.

Currently, the most human-intensive activity in the proposed approach is related to the definition of the oracle for each test case (assertions). Such an activity is conducted after the generation phase. Methods for the automatic recovery of (likely) invariants could be used to simplify this manual activity.

The study on the fault revealing capability of the generated test cases described in this paper is very preliminary. Additional experiments need be carried out, with a larger number of faults seeded into the class methods.

Moreover, *eToc* is still a research prototype that requires further engineering and extensions before being made publicly available. To reduce the manual work necessary for the production of the test cases, some support static analyses can be implemented, for example, to automatically add try-catch blocks in case of exceptions or to select the concrete classes that instantiate the parameters of interface or abstract type.

7. 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-2002*, New York, USA, July 2002.

[2] B. Baudry, F. Fleurey, J.-M. Jézéquel, and Y. Le Traon. Genes and bacteria for automatic test cases optimization in the .NET environment. In *Proc. of the 13th International Symposium on Software Reliability*

(ISSRE), pages 195–206, Annapolis, Maryland, USA, November 2002. IEEE Computer Society.

[3] B. Baudry, V. Le Hanh, J.-M. Jézéquel, and Y. Le Traon. Building trust into OO components using a genetic analogy. In *Proc. of the 11th International Symposium on Software Reliability (ISSRE)*, pages 4–14, San Jose, California, USA, October 2000. IEEE Computer Society.

[4] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133. ACM Press, July 2002.

[5] U. Buy, A. Orso, and M. Pezzè. Automated testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2000)*, Portland, OR, USA, August 2000.

[6] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, 2001.

[7] 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.

[8] M. J. Gallagher and V. L. Narasimhan. ADTEST: a test data generation suite for ada software systems. *IEEE Transactions on Software Engineering*, 23(8):473–484, 1997.

[9] N. Gupta, A. Mathur, and M. L. Soffa. Generating test data for branch coverage. In *Proc. of 15th IEEE International Conference on Automated Software Engineering (ASE'2000)*, Grenoble, France, September 2000. IEEE Computer Society.

[10] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. In *Proceedings of the 6th International Symposium on Foundations of Software Engineering (FSE)*, pages 232–244, Orlando, Florida, USA, November 1998. ACM Press.

[11] J. Henkel and A. Diwan. Discovering algebraic specifications from java classes. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP)*, pages 431–456. Springer, July 2003.

[12] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, 1990.

[13] B. Korel. Automated test data generation for programs with procedures. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 1996)*, pages 209–215. ACM Press, August 1996.

[14] V. Massol and T. Husted. *JUnit in Action*. Manning Publications Co., Greenwich, Connecticut, USA, 2003.

[15] C. C. Michael and G. McGraw. Automated software test data generation for complex programs. In *Proc. of IEEE International Conference on Automated Software Engineering (ASE'98)*, pages 136–146. IEEE Computer Society, 1998.

[16] R. Pargas, M. J. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Journal of Software Testing, Verifications, and Reliability*, 9:263–282, September 1999.

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