Semi-Automatic Search-based Test Generation

Yury Pavlov
Saarland University
Saarbrücken, Germany
Email: pavlov@st.cs.uni-saarland.de

Gordon Fraser
Saarland University
Saarbrücken, Germany
Email: fraser@cs.uni-saarland.de

***Abstract*—Search-based testing techniques can efficiently generate test data to achieve high code coverage. However, when the fitness function does not provide sufficient guidance, the search will only generate optimal results by chance. Yet, where the search algorithm struggles, a human tester with domain knowledge can often produce solutions easily. We therefore include the tester in the test generation process: When the search stagnates, the tester is given an opportunity to improve the current solution, and these improvements are fed back to the search. In particular, relevant problems occur often when generating tests for object-oriented languages, where test cases are sequences of method calls. Constructing complex objects through sequences of method calls is difficult, and often the traditional branch distance offers little guidance – yet for a human tester the same task is often trivial. In this paper, we present a semi-automatic test generation approach based on our search-based EVOSUITE tool, and evaluate the usefulness and potential on a set of example classes.**

***Keywords*-test case generation; search-based testing; manual testing**

I. INTRODUCTION

Software testing is an important technique to improve software quality, but finding good sets of test cases is a complex task. Search-based testing has been proposed as a possible technique to automatically produce test data for programs, resulting in test sets achieving high code coverage. Search-based testing uses meta-heuristic search techniques to find these input data, and as long as a fitness function provides guidance, this approach can be very efficient. However, the required level of guidance is not always available, and search operators might inhibit the generation of important genetic material. When this is the case, the search degenerates to a random search, and optimal solutions are only found by chance.

Although computers can be very efficient at tasks that are tedious for human users, in many cases where the search stagnates a human user would easily be able to improve the current solution. For example, Figure 1 shows a code snippet taken from the `ArgsParser` class from the DCParseArgs Sourceforge project. By looking at the code, it can easily be seen that to cover the target branch we need a string value for `key`, such that the command line handed to the `ArgsParser` as an array contains this key prepended with an argument indicator (`--`). However, the traditional branch distance measurement offers little guidance here, and so the search struggles to cover this branch.

```
public class ArgsParser {

 public SwitchArgument parseSwitchArgument(String key) {
  boolean isLongKey = (key.length() > 1);

  if (isLongKey) {
   String searchFor = LONG_ARGUMENT_INDICATOR+key;
   for (int i = 0; i < args.length; ++i) {
    if (innerArgs[i] != null) {
     if (innerArgs[i].equals(searchFor)) {
      innerArgs[i] = null;      ⇐[ Target branch ]
      return new SwitchArgument(i, key, true);
   // ...
 }
}
```

Figure 1. Code excerpt of the `ArgsParser` class: Covering the marked target branch is difficult for the EVOSUITE tool as it requires generating an array where one entry matches an input string concatenated with two dashes (`--`). EVOSUITE can cover the branch eventually, but needs a very large search budget to do so.

```
ArgsParser argsParser0 = new ArgsParser();
String string0 = "</xml>";      ⇐[ "--xml" ]
String[] stringArray0 = new String[4];
stringArray0[1] = string0;
argsParser0.setArgs(stringArray0);
argsParser.parseSwitchArgument(string0);      ⇐[ "xml" ]
```

Figure 2. Example test case generated by EVOSUITE (the string `</xml>` results from seeding), almost reaching the target branch, yet the last `if` statement evaluates to false. For a developer it is easy to spot that all that's necessary is to call the method with an option name, where the same name with two dashes pre-pended is added at any point in the array.

It is easy for the search algorithm to explore large parts of the program that a tester would consider tedious and dull work, but on the other hand it is easy for the tester to cover branches like the one in Figure 1 and very difficult for the search. Thus, in this paper we explore an approach that includes the user in the search process. Figure 3 presents this approach at a high level: First, the search algorithm aims to cover as much of the software under test as possible. Once the search stagnates, the best individual is presented to

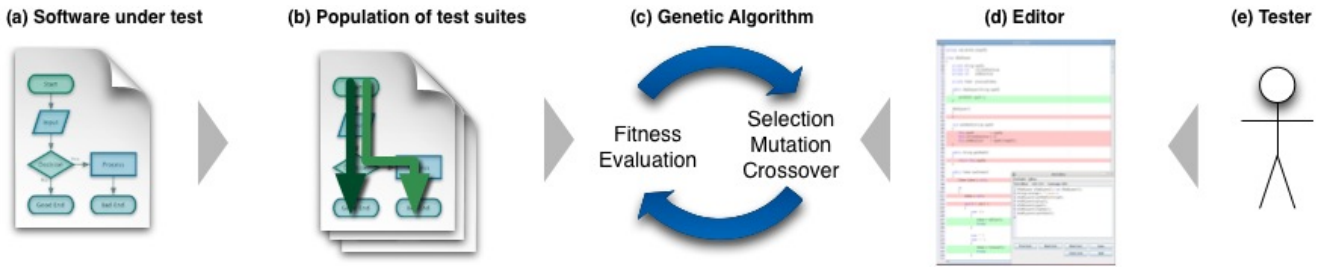

Figure 3. Our approach in a nutshell. For a given class under test (a) we generate an initial population of random test suites (b), which is successively evolved using a Genetic Algorithm (c). When the search stagnates, an editor window pops up (d), showing the current best test suite together with information on the coverage of the class under test. The tester (e) applies changes to this test suite as she/he sees fit, and the result is re-inserted into the population to let the Genetic Algorithm continue its search.

the user (for example, Figure 2 for the snippet in Figure 1) together with background information on where the search has problems. The user then improves the current solution, and once s/he is finished with this, the search continues with the solution edited by the user. If the user was helpful in escaping a difficult area in the search space, then the search algorithm will be able to cover more code after this manual editing phase.

Summarizing, the contributions of this paper are:

Semi-automatic unit test generation: We present an approach that integrates user-feedback into the genetic algorithm applied to generate unit tests for classes.

Tool integration: We have integrated our approach into the EVOSUITE tool, such that unit tests and the corresponding source code are presented to the user in a way that makes it easy to understand how to improve the current solution.

Evaluation: We have evaluated our approach on a set of 20 non-trivial classes selected randomly from the SF100 [4] Sourceforge project sample.

The evaluation demonstrates that our approach can significantly increase the coverage achieved with search-based testing, while at the same time it requires significantly less input from the user than during manual testing.

II. BACKGROUND

Software testing is an essential yet expensive activity in software development, therefore much research effort has been put into the question of how to automate it as much as possible. In the area of test data generation for code coverage, the current state of the art can at a high level be roughly divided into three main groups: variants of *random testing* (e.g., Randoop [9]), *dynamic symbolic execution* (e.g., CUTE [10]) and *search-based software testing* (e.g., [8]). A recent trend also goes towards combining the individual techniques (e.g., [7]).

Even though automated generation of test cases for structural coverage has received particular attention, for example in the case of object-oriented software (e.g., [11]), such testing techniques are still not widely adopted by practitioners. This is partially due to current limitations in these techniques (e.g., in terms of efficiency and applicability), and because many of the different parameters that influence search-based software testing (SBST) are not well understood. In order to identify the limitations of test generation, Xiao et al [12] described an approach to report problems that a testing tool encounters during test generation to the user. They identify external method calls and object creation as the major issues in test generation based on dynamic symbolic execution. Fraser and Arcuri [4] conducted a study on an unbiased sample of 100 open source projects taken from Sourceforge, showing that in practice environmental dependencies (e.g., file access) are a major inhibitor towards high code coverage, but also object creation and complex string-based calculations can be problematic for search-based testing.

When aspects of a Genetic Algorithm (GA) are difficult to compute automatically, human agents can be included in the search process. In Interactive GAs (e.g., [5]) human agents are used to evaluate the fitness of individuals where an automatic calculation is difficult, such as for example in fashion design. In Human GAs [6] all primary genetic operators are delegated to external human agents. The approach presented in this paper includes the user in the Genetic Algorithm to overcome problematic areas in the search space, thus automatic and manual steps alternate. The results of the manual editing are fed back into the population of the GA, where this new genetic material can boost the search. This is very much related to the idea of *seeding*, where previous related knowledge is exploited to help solve the testing problem at hand. For example, the EVOSUITE tool can seed the population of the GA using manually written test cases and constants extracted from the bytecode [3].

III. SEMI-AUTOMATIC SEARCH-BASED TESTING

We consider semi-automatic testing in the context of unit test generation, where test cases are sequences of method calls. In a unit testing scenario, there is always one dedicated target class, whose branches we aim to cover. The method sequences include calls to constructors and methods of this class, as well as any objects necessary as parameters. Consequently, test cases can be arbitrarily complex.

Algorithm 1 A genetic algorithm as used for search-based testing, extended with user feedback.

```
1  current_population ← generate random population
2  repeat
3    Z ← elite of current_population
4    while |Z| ≠ current_population do
5      P1, P2 ← select parents from current_population
6      if crossover probability then
7        O1, O2 ← crossover P1, P2
8      else
9        O1, O2 ← P1, P2
10     end if
11     mutate O1 and O2
12     fP = min(fitness(P1), fitness(P2))
13     fO = min(fitness(O1), fitness(O2))
14     if fO ≤ fP then
15       Z ← Z ∪ {O1, O2}
16     else
17       Z ← Z ∪ {P1, P2}
18     end if
19   end while
20   current_population ← Z
21   if search stagnated then
22     S' ← copy best individual
23     Let user edit S
24     Z ← Z ∪ S'
25     Remove worst individual from Z
26   end if
27 until solution found or maximum resources spent
```

A. Technique

A popular meta-heuristic search technique often applied in search-based technique is a Genetic Algorithm (GA). Algorithm 1 shows a typical GA: First, an initial population of candidate solutions is produced randomly. Then, selection, crossover, and mutation are applied successively, until the next generation is finished. This process is iterated until either an optimal solution has been found, or another stopping condition has been met (e.g., out of time). The probability of an individual being selected for reproduction is related to its fitness with respect to the optimization problem, such that gradually the population improves.

Our aim is to produce test sets that achieve high code coverage. For this, we apply the whole test suite generation approach used in our EvoSuite [1], [2] tool. In this approach, a chromosome of the search is a set of test cases, where each test case is a sequence of method calls. Crossover exchanges tests between two parent sets, while mutation inserts, deletes, or changes individual test cases. Mutating test cases involves insertion, change, and deletion of statements. The fitness function measures the minimal branch distances for each branch in the target class, and basically calculates the sum. If the sum of branch distances is 0, this means that all branches have been covered. For more details about this approach, we refer to [1].

As long as the fitness improves, there is no need to ask the user for input. However, once the search stagnates on a sub-optimal solution, we ask the user to improve the best solution. There are different ways to decide when the search has stagnated. We use two parameters to determine this: Number of iterations I, and minimal fitness improvement δ. If over I generations the fitness improvement is smaller than δ then we request user input.

As it is not feasible to request the user to treat the whole population, we only present the current best individual; i.e., the user gets to see the best test suite To prevent that the user adversely influences the integrity of the population, the user modifies only a copy of this individual. Once the user is finished with his modifications, the resulting individual is re-inserted into the population, and to keep the population size constant the worst individual is removed instead. If the user does not make many changes, then this might slightly reduce the diversity in the population; however, as user interaction should occur as infrequently as possible this should not be a problem.

B. Tool Implementation

We have implemented this technique as an extension to our EvoSuite tool. EvoSuite considers individual classes, and candidate solutions are sets of test cases. The number of tests in a set as well as their length are variable. EvoSuite uses a secondary objective that favors smaller test sets over longer ones, yet when taking individuals out of the population during the search it is likely that they contain a lot of noise (e.g., unnecessary test cases, unnecessary statements in the test cases). As the user has to make sense of these test cases in order to modify them, we apply an optimization to the test set that is intended to improve the readability of the test cases: For each covered branch in the class under test we select on test case out of the test set that covers this branch, and generate a minimized sequence of method calls that still covers the branch. Typically, the resulting test cases are very short, in the order of 2-3 statements.

Figure 5 shows the editor window that the user gets to see: The editor allows modification of all tests just like a regular IDE, and uses standard editing features like syntax highlighting. Tests can be edited directly or duplicated, such that the modifications can be done on copies, new tests can be inserted, and existing tests can be deleted.

As a user might not immediately be aware of the consequences of each of his/her actions, the modifications might result in an individual of worse fitness than the current best individual of the search, even if new branches are covered. If this would happen, the new genetic material introduced by the user might be lost within a few generations. To prevent

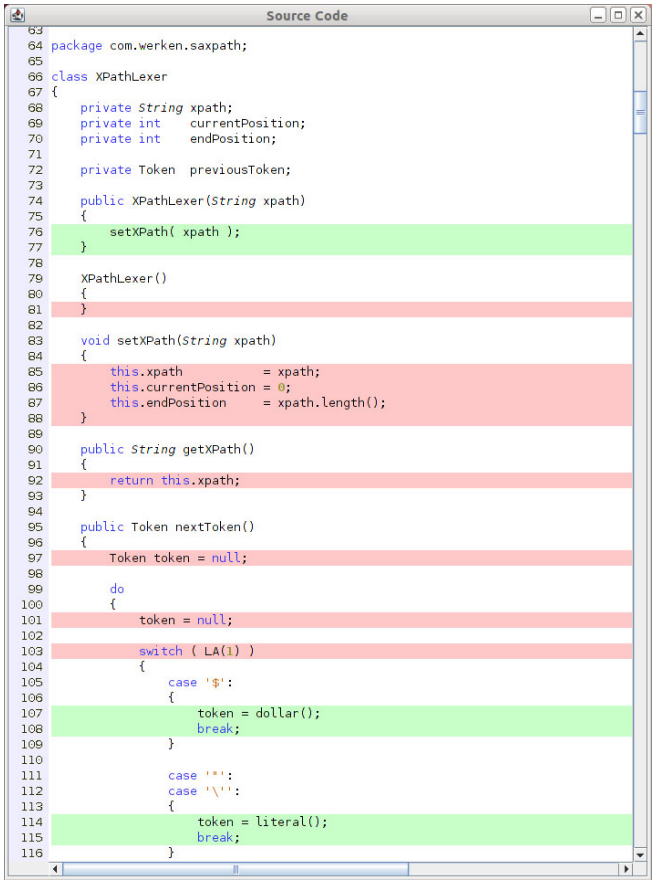

Figure 4. Source code GUI: Coverage of the test suite as well as the coverage of the currently edited test case are highlighted.

this from happening, we therefore check the resulting test set before re-insertion in the population: For each branch that was covered before the modifications but is no longer covered we insert a test from the original test set that covers this branch. This way, after re-insertion, the best individual is guaranteed to be the modified test set.

To support the user further in his/her modifications, EvoSuite displays an editor window showing the source code of the class under test (see Figure 4). In this editor window, lines and branches covered by the test set that is edited and those covered by the currently edited test case are both highlighted separately. This allows the user to quickly spot which branches need to be covered, and what effects the current test has.

IV. Evaluation

In order to determine whether semi-automatic testing is useful in practice, we performed an experiment to answer the following two research questions:

RQ1: How much can semi-automatic testing increase coverage over fully-automatic test generation?

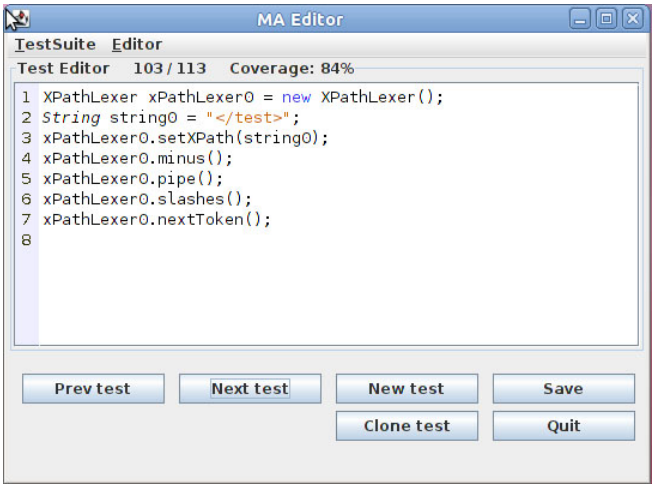

Figure 5. Text editor GUI: The editor parses user input, and converts it to EvoSuite's internal format.

RQ2: How much does semi-automatic testing reduce the effort over manual testing?

A. Experimental Setup

Ideally, RQ1 and RQ2 need to be answered by a controlled experiment, such that variations in the qualification, background knowledge, programming skills, etc. can be factored out, while measuring effectiveness through time. As an initial experiment, we approximated the human effort by counting the statements in manually written test sets and comparing them to the edits that need to be done during semi-automatic testing. This measurement avoids the threat to validity that would be caused by measuring the time, as a significant share of the time is always spent for program comprehension, yet only needs to be done once for manual and semi-automatic testing. In both cases we continued testing until all feasible branches were covered.

In our experimental setup, we applied semi-automatic testing to a set of classes ourselves. We configured EvoSuite with a maximum number of $1,000,000$ executed statements or a maximum of 10 minutes of computation time for fitness evaluations; the time for manual editing is not included in the 10 minute count. I was set to 300, and δ was set to $0,01$. For all other parameters, we used the default values provided by EvoSuite. For each class, we analyzed the existing JUnit test cases, and improved them until no further coverage could be achieved – this also involved determining the infeasible branches of the classes under test.

The selection of case study classes needs to be non-trivial, such that EvoSuite does not achieve 100% coverage within the given search budget. On the other hand, we did not want to include classes that have environmental dependencies (e.g., they might need to read from files or access the network). Although it would be possible to treat these cases also with semi-automatic test generation, in this study we

Table I
STUDY SUBJECTS

Id	Project	Class	#Lines[1]	#Branches
1	Tullibee	Order	191	135
2	OMJState	StringMatchesGuardCondition	27	5
3	Tullibee	EWrapperMsgGenerator	366	67
4	SaxPath	XPathLexer	784	484
5	OpenHRE	SAXEvents2HL7Impl	181	70
6	Tullibee	Util	50	33
7	WheelWebTool	Validations	44	18
8	GAE App Manager	Main	47	6
9	WheelWebTool	ProjectCreator	60	5
10	Heal	UserInfoBean	63	66
11	Celwars2009	BezierSpline	51	15
12	JCVI JavaCommon	FollowData	34	6
13	Asphodel	SimpleAnalyzer	41	7
14	DCParseArgs	ArgsParser	139	80
15	Corina	Raw2Pack	57	4
16	TemplateIT	FormulaUtil	65	28
17	Jiggler	GDilate	95	27
18	TemplateIT	OpMatcher	121	45
19	JIPA	Label	33	11
20	OMJState	IntegerGreaterThanGuardCondition	24	4

want to focus on providing user input only of a type that can also be represented by EVOSUITE's test representation, such that EVOSUITE could in theory also derive the same solution, if given enough time.

To avoid a bias in our case study class selection, we therefore randomly chose 20 classes out of the SF100 corpus of Java projects randomly selected from Sourceforge [4]. We considered all classes that had caused no unsafe actions during test generation initially, and for which EVOSUITE achieved more than 15% but less than 100% branch coverage. This resulted in the selection of classes shown in Table I.

B. Results

Table II lists the coverage achieved by EVOSUITE by itself, the intervals of improvement after manual editing, and the final coverage. The column on semi-automatic testing lists ranges, where $x-y$ means that manual editing increased the coverage from EVOSUITE's result up to $x\%$ coverage, and then EVOSUITE managed to increase the coverage further up to $y\%$. If there is more than one range listed in this column, this means that there was more than one manual editing phase. The last column shows the final level of coverage achieved through semi-automatic and through manual testing (in both cases we continued testing until all feasible branches were covered). For purely manual testing, we used the existing JUnit test suites as a starting point, and wrote additional tests until all remaining branches were either covered, or revealed as infeasible. Note that this column does not show how much coverage is actually achieved by the developers of the individual classes, as our experiment setup does not aim at comparing the coverage achieved by manual testing with other approaches.

[1]Non-commenting lines of source code, calculated with CLOC (http://cloc.sourceforge.net)

Table II
BRANCH COVERAGE ACHIEVED

Id	EVOSUITE	EVOSUITE after Manual Editing	Manual / Final
1	77%	80%-82%, 84%-85%	100%
2	60%	90%-100%	100%
3	79%	94%-98%	100%
4	85%	89%-92%	100%
5	60%	74%-87%, 90%-91%	97%
6	63%	86%-96%	100%
7	72%	-	94%
8	16%	-	33%
9	20%	-	100%
10	80%	90%-100%	100%
11	66%	-	100%
12	33%	33%-100%	100%
13	57%	85%-100%	100%
14	86%	93%-96%	97%
15	25%	-	50%
16	82%	82%-85%, 96%-100%	100%
17	96%	-	100%
18	73%	84%-86%	86%
19	45%	-	100%
20	50%	-	100%

Table III summarizes the coverage increase achieved over fully automatic EVOSUITE. In all cases there is a clear increase in coverage, which demonstrates that semi-automatic testing is suitable to improve coverage when search-based test generation cannot achieve optimal results due to constraints on the search budget, or limitations of the search operators.

In our experiments, semi-automatic test generation increased branch coverage by 34.63%.

Table IV lists details of the edits that were performed during the manual modification steps. The column labeled

Table III
BRANCHES COVERED BY EVOSUITE WITHOUT AND WITH SEMI-AUTOMATIC TESTING.

Id	EVOSUITE	Semi-Automatic	Coverage Increase
1	104	135	22.96%
2	3	5	40.00%
3	53	67	20.90%
4	411	484	15.08%
5	42	68	38.24%
6	21	33	36.36%
7	13	17	23.53%
8	1	2	50.00%
9	1	5	80.00%
10	53	66	19.70%
11	10	15	33.33%
12	2	6	66.67%
13	4	7	42.86%
14	69	78	11.54%
15	1	2	50.00%
16	23	28	17.86%
17	26	27	3.70%
18	33	39	15.38%
19	5	11	54.55%
20	2	4	50.00%
Average			**34.63%**

Table IV
NUMBER OF EDITED STATEMENTS

Id	Added	Modified	Manual	Reduction
1	4	28	212	84.91%
2	4	0	10	60.00%
3	7	2	53	83.02%
4	1	52	93	43.01%
5	8	3	55	80.00%
6	5	3	32	75.00%
7	0	3	16	81.25%
8	1	0	2	50.00%
9	4	1	10	50.00%
10	0	5	36	86.11%
11	0	2	4	50.00%
12	4	0	6	33.33%
13	0	1	4	75.00%
14	4	1	69	92.75%
15	1	0	2	50.00%
16	0	2	11	81.82%
17	0	1	9	88.89%
18	0	3	35	91.43%
19	2	1	4	25.00%
20	6	1	9	22.22%
Average				**65.10%**

"Added" lists the number of statements that were newly added to the test set, while "Modified" shows how often existing statements produced by EVOSUITE were simply modified. For example, this is often the case when EVOSUITE produces a random string that needs to be adapted to some particular expected values. To compare these edits with the effort involved in manual testing, we measure the number of statements in the manual test sets, which is shown in the fourth column. Finally, the last column shows the reduction compared to the numbers of instructions written in the manual test suites for the same level of coverage. For this comparison, we also count each modification of a value during semi-automatic testing like a new line insertion. In all cases the effort is greatly reduced.

```
public boolean evaluate(Object o) {
  boolean rc = false;
  try {
    Vector params = ((Event)o).getParameters();
    String str = (String)params.elementAt(0);
    if(Value.equals(str)) {
      rc = true;     ⇐ Target branch
    }
  } catch(ClassCastException ex) {
    ex.printStackTrace();
  }
  return rc;
}
```

Figure 6. Excerpt of the `StringMatchesGuardCondition` class. EVOSUITE has problems reaching the target branch, as it just tries to match the method signature when calling `evaluate`, and thus an `Event` object is only passed in by chance. The tests produced by EVOSUITE thus mostly raise a `ClassCastException`.

```
String s0 = "";
StringMatchesGuardCondition smgc0 = new
      StringMatchesGuardCondition(s0);
smgc0.getValue();
smgc0.evaluate(s0);
```

Figure 7. A test case produced by EVOSUITE to exercise `evaluate`. As a `String` object is passed in, the `ClassCastException` is triggered.

In our experiments, semi-automatic test generation reduced the number of test code statements written for the same coverage by 65.10% over manual testing.

C. Problems in Object-Oriented Test Generation

A closer look at the branches that EVOSUITE was not able to cover without human intervention demonstrates where search-based testing currently has problems, and illustrates how semi-automatic testing works in practice. For example, consider the code excerpt in Figure 6: The signature of the method describes that an instance of `Object` is requested. EVOSUITE will blindly try to match this method with any object that is assignable to `Object` – which, alas, all objects are. The chances of selecting a required `Event` class are very small. In fact, this problem is quite common in Java, as the Java compiler strips away all type information from Java Generics, such that any generic parameters or return

```
String s0 = "test";
StringMatchesGuardCondition smgc0 = new
      StringMatchesGuardCondition(s0);
smgc0.getValue();
Vector v = new Vector();
v.add(s0);
Event e = new Event(s0, v, new Object());
smgc0.evaluate(e);
```

Figure 8. Manual editing of the test case in Figure 7 resulted in this test case. Seeing the source code, it is obvious we need to create an `Event` object that contains the string `s0`.

```
  // ...
  p(0, 0, controlPoints_, spline, 0);
  // ...
}

private void p(int i, double t, double cp[], double spline[], int
      index) {
  double x = 0.0;
  double y = 0.0;
  double z = 0.0;

  int k = i;
  for(int j = 0; j <= 3; j++) {
    double b = blend(j, t);
    x += b * cp[k++];
    y += b * cp[k++];
    z += b * cp[k++];
  }
  spline[index + 0] = x;      ⇐ Target
  spline[index + 1] = y;
  spline[index + 2] = z;
}
```

Figure 9. Excerpt of the `BezierSpline` class. EVOSUITE has problems reaching the target branch, as the array `cp` needs to match the number of elements represented by `i`, and `i` needs to be positive. For this, branch coverage does not provide sufficient guidance. Note also that the target branch is contained in a private method, and cannot be directly called during testing.

```
double[] doubleA0 = new double[9];
int int0 = −21;
BezierSpline bezierSpline0 = new BezierSpline(doubleA0, int0);
bezierSpline0.generate();
```

Figure 10. A test case produced by EVOSUITE for the `BezierSpline` class. Method `p` is called indirectly by `generate` with an `i` of -21, which causes an exception, and an array of length 9.

values in bytecode are seen as `Object`. Manual editing can overcome this problem very easily: Figure 7 shows an example test case produced by EVOSUITE which can be extended to cover the target branch, by simply creating a suitable `Event` object, as shown in Figure 8. Once this object is in the population, EVOSUITE can easily cover all dependent branches. In theory, this problem could also be overcome by considering type constraints caused by casts, and we are working to extend EVOSUITE in this direction.

Figure 9 shows another example where EVOSUITE has problems reaching a target branch. The target is contained in a private method, which takes several parameters, including an array and an index into this array. The target branch

```
double[] doubleA0 = new double[12];
int int0 = 1;
BezierSpline bezierSpline0 = new BezierSpline(doubleA0, int0);
bezierSpline0.generate();
```

Figure 11. Manual editing of the test case from Figure 10 fixing the array size and the index can easily achieve full coverage of `p`.

```
// ...
private static void offsetRelativePtg(Ptg ptg, int roff, int coff)
      {
 // ...
 else if (ptg instanceof AreaPtg) {
  AreaPtg aptg = (AreaPtg) ptg;    ⇐ Target
  if (roff != 0) {
    if (aptg.isFirstRowRelative()) {
      aptg.setFirstRow(aptg.getFirstRow() + roff);
    }
    if (aptg.isLastRowRelative()) {
      aptg.setLastRow(aptg.getLastRow() + roff);
    }
  }
  // ...
 }
 // ...
```

Figure 12. Excerpt of the `FormulaUtil` class. In order to cover these branches, EVOSUITE needs to create string inputs that represent row and column ranges in a spreadsheet (e.g. A1:A10), such that a `AreaPtg` object is constructed from it.

```
HSSFWorkbook wb0 = null;
String string0 = "9";
int int0 = 20;
FormulaUtil.offsetRelativeReferences(wb0, string0, int0, int0);
```

Figure 13. A test case produced by EVOSUITE for the `FormulaUtil` class. The string does not represent a valid cell range.

can only be covered if EVOSUITE produces a combination of an array of size greater 11 and a positive index, and if this combination is set accordingly through the constructor of the `BezierSpline` class. Figure 10 shows a test case produced by EVOSUITE: As the index is negative, the target method throws an exception before the target branch is covered. This can be easily fixed, and does not even require addition of new statements. As shown in Figure 11, we simply need to change the array size and the index in order to cover the target branch.

Finally, Figure 12 shows an example where a branch in a private method indirectly depends on a string value passed and processed at other places in the code: The string needs to represent a cell range, such that an `AreaPtg` object is constructed. To cover this branches, EVOSUITE usually needs to be run for a very long time as the branch using `instanceof` offers no guidance. Yet, fixing the tests produced by EVOSUITE (Figure 13) is a matter of changing a simple string (Figure 14). Once the branch is covered, EVOSUITE can cover all branches control dependent on it easily by itself.

V. THREATS TO VALIDITY

This paper presents a preliminary study on semi-automatic testing, and is thus suspect to a number of threats to validity. Threats to *construct validity* exist because we measured the increase in branch coverage compared to purely automatic

```
HSSFWorkbook wb0 = null;
String string0 = "A9:B9";
int int0 = 20;
FormulaUtil.offsetRelativeReferences(wb0, string0, int0, int0);
```

Figure 14. To cover more branches in `FormulaUtil` the manual editing of this test case simply changes the string to a valid cell range.

testing, as well as the reduction in the number of statements written by a tester. This measurement does not take into account how familiarity with the unit under test, experience, or program comprehension affect the overall usability of the approach. This might be better reflected by measuring the time; for example, tests produced by EVOSUITE might be confusing, slowing down the tester during manual testing phases. Such effects can only be observed with user studies. Threats to *internal validity* might come from how the empirical study was carried out. To reduce the probability of having faults in our testing framework, it has been carefully tested. Threats to *external validity* regard the generalization to other types of software, and are common to any empirical analysis. To avoid a bias in the selection of classes, we selected them randomly out of the SF100 corpus [4], filtering out by several criteria (minimum and maximum coverage). Yet, larger experiments will be required in the future.

VI. CONCLUSIONS

In this paper, we have presented an approach to include the tester during search-based test generation in a unit testing scenario for object-oriented software. We have extended our EVOSUITE tool to show an editor window presenting to the user a preprocessed version of the current best individual in the search population if the search stagnates. The user can modify this test suite in any desirable way, and the resulting test suite is fed back into the population of the Genetic Algorithm.

Our initial set of experiments demonstrated the usefulness of this approach; coverage is clearly higher than in fully automatic testing, yet the effort is significantly smaller than in manual testing. However, there is much potential for future work:

- We used default values for the parameters I and δ, but a sensitivity analysis of these parameters is necessary to find out the best values.
- Evaluation needs to be done in terms of user studies to find out how developers and testers can cope with semi-automatic testing in practice.
- Many of the problems that semi-automatic testing can solve are due to limitations in the search operators or fitness function. When such problems are identified, it may in many cases be possible to improve the search such that higher coverage can be achieved automatically, further reducing the manual input necessary.
- If the limitations of the tool are precisely known, the search might not want to wait until stagnatation, but could ask questions much earlier ("How do I produce a value for this method such that we can reach this branch?").
- Semi-automatic testing may be very well suited to overcome the problem of environmental dependencies [4]. In fact, three of the classes selected in our experiments contained unexpected dependencies on files, which were easily overcome with semi-automatic testing.

More information about EVOSUITE is available at

`http://www.evosuite.org/`

Acknowledgments. This work was supported by grant Ze509/4-1 from Deutsche Forschungsgemeinschaft and a Google Focused Research Award on "Test Amplification".

REFERENCES

[1] G. Fraser and A. Arcuri. Evolutionary generation of whole test suites. In *International Conference On Quality Software (QSIC)*, pages 31–40, Los Alamitos, CA, USA, 2011. IEEE Computer Society.

[2] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, 2011.

[3] G. Fraser and A. Arcuri. The seed is strong: Seeding strategies in search-based software testing. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2012. To appear.

[4] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2012. To appear.

[5] H.-S. Kim and S.-B. Cho. Application of interactive genetic algorithm to fashion design. *Engineering Applications of Artificial Intelligence*, 13(6):635 – 644, 2000.

[6] A. Kosorukoff. Human based genetic algorithm. volume 5, pages 3464–3469, 2001.

[7] J. Malburg and G. Fraser. Combining search-based and constraint-based testing. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, 2011.

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

[9] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ACM/IEEE Int. Conference on Software Engineering (ICSE)*, pages 75–84, 2007.

[10] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE-13: Proc. of the 10th European Software Engineering Conf. held jointly with 13th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering*, pages 263–272. ACM, 2005.

[11] P. Tonella. Evolutionary testing of classes. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*, pages 119–128, 2004.

[12] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux. Precise identification of problems for structural test generation. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 611–620, New York, NY, USA, 2011. ACM.