Human-Centric Test Generation

Adriana Libório
Federal University of Pernambuco
Informatics Center, Recife-PE, Brazil
alfl@cin.ufpe.br

***Abstract*—Despite the tremendous advances observed over the recent years in automated test generation, existing techniques are far from achieving their goal of fulfilling some informed coverage criteria. That happens because the search-space that generators explore are often intractable. This paper presents one approach to involve the human actively in the test generation process. Intuitively, humans can easily solve problems that a machine encounters difficulty, and conversely. Our goal is to facilitate the interaction between the human and the tool for the particular purpose of test generation. Our solution builds on the infrastructure of the search-based EvoSuite test-generator. We evaluate our approach on a well-known benchmark (SF100); results show that by sparing the search from hard-to-solve problems, the search is able to improve its performance and in most cases solve subproblems on its own.**

I. Introduction

Testing is important. As long as humans continue to participate in software development, errors will likely continue to be introduced in code, and software testing will remain an important activity in software engineering. In industry, testing is certainly today the most popular technique for verification and validation of software.

Unfortunately, testing has its limitations. It is incomplete and can consume significant human energy. Automation can help improve testing. Previous test-automation techniques showed some promising results in some evaluation settings when using controlled code. However, a recent study [1] shows that those techniques do not work well "out of the box", on real open-source projects: the study involved 100 randomly-selected real applications and indicated that effectiveness of several test-generation techniques proposed in the literature have been overestimated. The study found that, when considering programs that contain a significant amount of dependencies with the testing environment, the effectiveness of test-generation techniques is significantly lower than what has been advertised. In particular, the study found that test generation performs poorly for programs with dependencies on databases, files, and other sub-systems. Unfortunately, the study also found that these programs seem to dominate in practice. In a different study, involving 4 large applications, Xiao *et al.* [2] also report similar difficulties that test-generation techniques face. These two empirical studies applied different testing techniques (e.g., random, evolutionary, and symbolic test generation) but reported similar limitations. Their results suggest that these limitations correspond to general challenges of search as opposed to particular deficiencies of existing techniques.

This paper exploits one way to enable humans and test generators to interact, synergistically leveraging the power of each other. More precisely, testers can provide guidance to the search on the basis of the difficulties that the test-generator faces. The main challenge is to identify the proper interfaces between the user and the tool. This paper proposes a semi-automatic approach for test generation that builds on the `EvoSuite` [3] test generator.

Figure 1 shows a code excerpt of `ArgsParser` class. To cover the target branch it is necessary to generate a string starting with two dashes ("- -"). `EvoSuite` can only cover such a branch when given a very large time budget.

```
public class ArgsParser { ...
 String LONG_ARGUMENT_INDICATOR = "--";
 ...
 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);
   ...
 }...}
```

Fig. 1. Excerpt of `ArgsParser` class.

Figure 2 shows an example test sequence that `EvoSuite` generates. This sequence comes close to covering the target branch. However, the last `if` statement evaluates to `false`. Based on the content of `LONG_ARGUMENT_INDICATOR` it is easy for a developer to realize that all that's necessary to achieve a `true` evaluation on this last `if` statement is to call the method with the same name but with two dashes prepended to it.

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

Fig. 2. Example test sequence that `EvoSuite` generates. The side comments show the String arguments needed to cover the target branch indicated in Figure 1. The two dashes are required due to the value of `LONG_ARGUMENT_INDICATOR` in Figure 1.

Fraser and Arcuri [1] reported some major problems faced by test-generation techniques, in particular for the

EvoSuite. Complex string handling, Java generics, dynamic type handling and branches in exception handling code are among the reasons why a test-generation technique cannot always achieve a high coverage. For example if there is a call for a String.equals method, EvoSuite won't always be able to create a test case that would generate both true and false return calls for this method in a short period of time. Another limitation of the EvoSuite occurs when the class requires I/O for reading/writing a file or even for network permission checks. There are branches which depends on or sometimes even create files causing the EvoSuite to generate several files with random names. Classes with graphical user interface also have problems when used in a test-generation technique. A large amount of windows being opened at the same time requiring user inputs, together with a large amount of thread manipulation makes this scenario close to impossible to handle with a test-generation technique.

Our approach helps the tester to identify hard-to-solve branches and generate tests that cover them. We build on search-based test generators like Randoop [4] and EvoSuite [3]. We say that the search hits *stagnation*, when coverage does not improve for a certain (informed) time interval. A stagnation in the search reveals a *stagnation frontier*, which corresponds to the set of branches that the search is about to solve (but could not). Note that it is possible that such set is small and that hard-to-solve branches dominate not only many easy to solve branches but also some other hard-to-solve ones. Our approach to test generation proceeds in an interactive way: the user collaborates with the tool for a number of times corresponding to the number of stagnation hits. When interruption occurs, the tester adjusts tests that the tool informs with the goal to cover the branches in the stagnation frontier. We conjecture that the number of stagnation hits is small and so is the number of branches to cover in each stagnation frontier.

This paper makes the following contributions:

Idea: We present a semi-automatic test generation technique that brings the human to the loop. The human assumes an active role of assisting the test generator in improving coverage.

Implementation: Our approach is supported by a tool, which includes an environment for observing search progress, stagnation frontiers and their branches, and refining tests and evaluating whether they cover frontier branches. Our tool builds on the `EvoSuite` test-generator.

Experiments: We evaluate our approach using the SF100 benchmark. We want to observe how often and how quickly stagnation occurs as well as the number of branches in each frontier a stagnation reveals. Results indicate that search stagnates really quickly and by solving a frontier, the search is able to improve coverage even more than just that branch.

II. Background and Related Work

Many automatic generation techniques have been proposed, such as Randoop [4], which is based on random approach to generate test inputs. Because it is open-source, a lot of other test generators and extensions were created based on Randoop. In dynamic symbolic execution, Microsoft developed Pex [5] as a test generator for their language C#, and for `Java` we have jCute [6]. In test generation based on search, EvoSuite [3] is a state of the art with its evolutionary search to create test cases for Java applications. In this project we focus our approach as an extension of EvoSuite.

Different static metrics to predict the capability of test-generators on programs have been studied [7], [8]. They can be useful to give rough explanations for the reasons why test generators did not work. Previous research has also pointed to different factors that influence test-generator capability [9]–[11]. All these results suggest new directions for improving existing test-generator tools.

Despite the remarkable advances observed in previous years, recent studies indicate that there is still a lot to improve in automated test-generation research [1], [2]. Xiao *et al.* [2] and then Fraser and Arcuri [1] report similar problems in automated test generation which are inherent to the challenging search spaces that generators need to explore. They showed that state-of-the-art techniques and tools have not been able to achieve high coverage when it comes to real-life application. For example, applications that involve I/O and environmental dependencies [1]. The focus of this paper is to improve human-machine collaboration with the goal to improve effectiveness of automated test generation and therefore contribute to reduce the gap that exists between testing research and practice.

Test Sequence Generation. Pavlov and Fraser have recently investigated human guidance in search-based testing [12]. Their work builds on the idea of human-based genetic algorithms [13]. The approach taken by Pavlov and Fraser was to stop the search when it stagnates, i.e., coverage scores observed during the search saturate. In such scenario, the tool asks the tester to augment the test suite with informed test cases. Their results suggest that human-interaction is one concrete way to deal with the problem of environmental dependencies [1]. Our approach focus on problems that need to be solved in that moment, as the next to cover branch. We study how to simplify, minimize the user effort and how to use user feedback to solve other problems. Ning and Kim proposed PAT [14] to improve test-generation with the user help. In contrast to the work of Pavlov and Fraser, PAT uses symbolic execution to produce object and primitive-type constraints that existing off-the-shelf constraint solvers cannot handle. PAT fragments and presents these constraints for the user to solve.

Test Oracle Generation. Since machine limitations, CrowdSourcing has began to be studied as a way to be able to achieve goals that computation only can't. CrowdSourcing is a recently popular technique to automate computation that cannot be performed by machines, but only by humans. In this very recent study [15], Pastore introduces an approach to solve test generation problems using this technique by simplifying these problems into subproblems that mostly unqualified crowd are able to fix them.

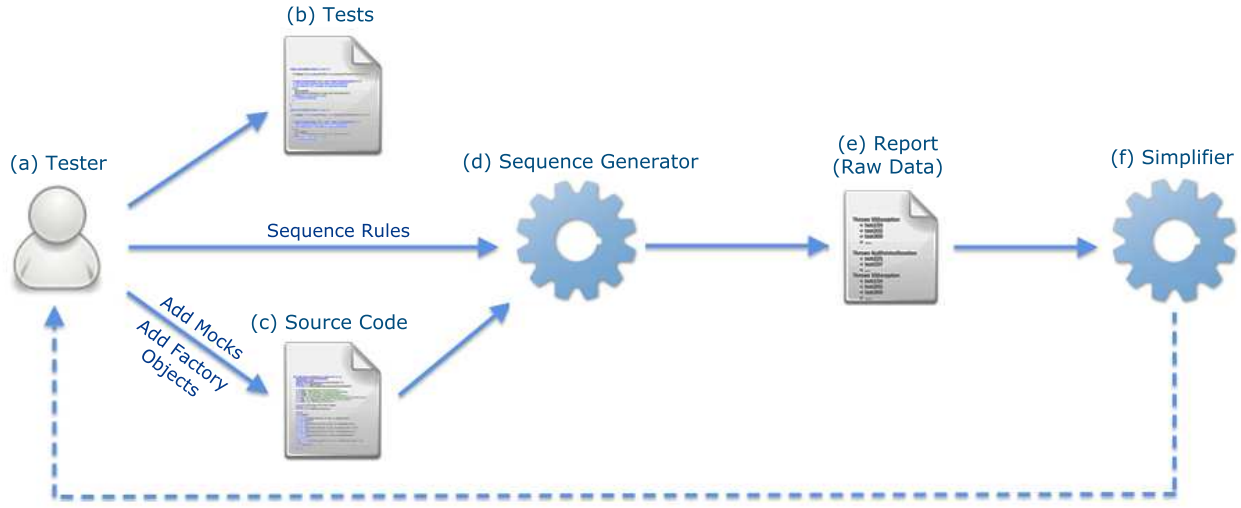

Fig. 3. Test generation workflow. This diagram shows our approach interaction loop with the user and between components.

III. Technique

Our technique uses human-feedback to improve automated test generation. Figure 3 illustrates the test generation process. The feedback loop continues until the tester is satisfied with the quality of the generated test suite. The user (a) can either create new tests or improve existing ones (b). These tests are the seeds to the (c) sequence generator (d). The simplifier (f) processes the data collected during the search up to a stagnation bound and communicates to the user the branches in the frontier and the tests that got close to cover them. The framework provides a user-interface to assist the user with the elaboration of tests to cover branches in the frontier. We detail below each of the individual processes in this workflow.

A. Sequence Generator

The goal of this component is to maximize branch coverage of an input class under test. The sequence generator builds the scope of the search dynamically. When the search makes no progress for a certain (informed) amount of time it stops and all collected data is passed to the simplifier. In principle, our approach is general to use any randomized test generator. We used the `EvoSuite` generator.

EvoSuite. Our approach builds on the EvoSuite evolutionary test generator [16]. EvoSuite uses genetic algorithm to perform search for test sequences. Each individual in the search population is a test suite (a set of test cases). So there is no distinction between unit and integration testing in this context. A test case is a sequence of method calls, however, the user can provide annotations to be used as test oracles. The method call sequences can be as complex as needed to set right objects instances, change attributes, etc. We refer to the *size of a test case* as the number of statements it has. In evolutionary test generation [1], [3], [9]–[12] it is possible that the search gets stuck trying to cover particular hard-to-cover branches. This can happen because the size of the search space can be large, making the search to virtually stop or to make progress at an unacceptable rate. `EvoSuite` uses a standard notion of *branch distance* to evaluate fitness [16].

EvoSuite adaptation. We adapted search to keep a more global state of the genetic algorithm. Since in `EvoSuite` individuals of the population are suites, if a test case in another individual covers a branch not covered yet, but this is not part of the best individual, this improvement is only improves that individual fitness, but not the whole population coverage. We added this information in a global state where every improvement in the population is used and added back as a better individual to search population. This way, we can achieve better coverage result faster, and then stagnate (if necessary) with branched in the frontier that search really didn't find a solution.

B. Simplifier

The tester is a key participant in human-centric test generation. It is therefore crucial that an interactive environment reports useful information, e.g., a small set of facts that reveals the difficulties that the test generator faces. In this component, we try to formulate various simple questions about execution behavior to simplify the answers to the user. One can formulate a problem as: find groups of generated test sequences that fail to cover particular conditional branches and report them to the user in a compact manner.

The main goal of the simplifier, is to minimize the information that we show to the user. For example, when search stagnates, it stagnated with a frontier of branches that weren't covered yet. We define frontier branches as those branches that were not covered yet but their dependencies have been, so they are the next branches to be covered. So, from the search data that we saved, we have the test cases that get the closest to cover that branch. However, that test case can be as large as the maximum number of statements in a test case, and the first thing necessary to do is to minimize this test case, to leave only the minimum of statements that are necessary to keep the same distance to that branch. Figure 5 shows the test case editor screen

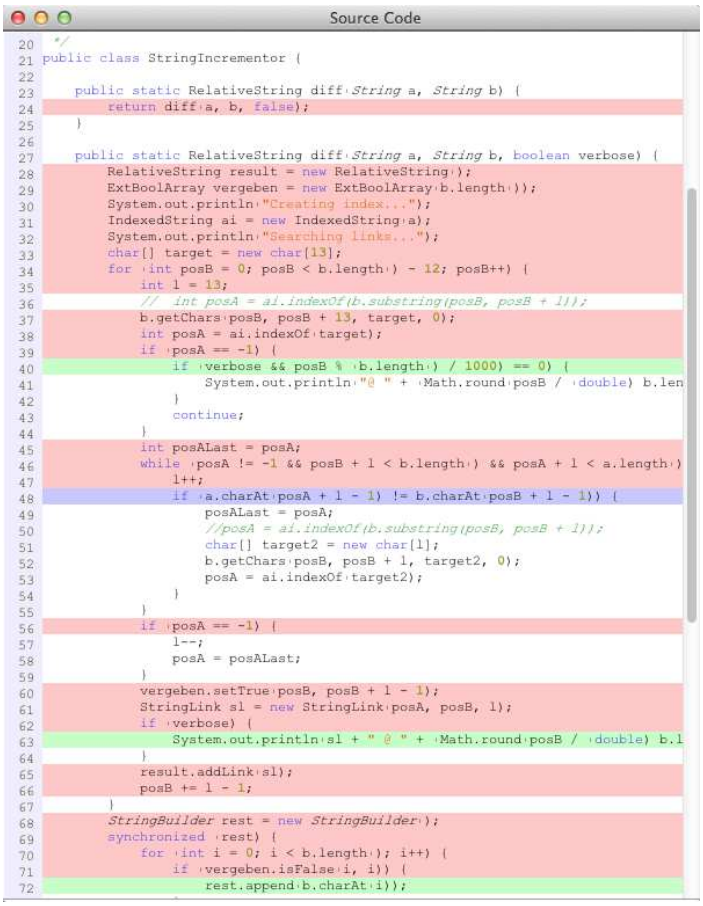

Fig. 4. Source code GUI: The test cases coverage is highlighted in the source code. Statements highlighted in blue are the target branch of given test case, red are the statements being executed in the given test case and green are the ones that were covered by other test cases.

that the user can use to edit the minimized test case that targets the blue colored statement in Figure 4. For each of the frontier branches, we show the best test case for it and in the source code screen 4 we highlight statements that were covered in that test case (red) and by the other test cases (green). The simplifier can be formulated as another search problem. Conceptually, the goal of the simplifier is to report to the user a quality summary of the test generator difficulties.

C. User-Interface (UI)

Once the stagnation point is reached, we need to ask feedback to the user. For this purpose, we show to the user the GUI shown in Figures 4 and 5. At this moment, the user can visualize the best test case to achieve the target branch (which is displayed in blue for him) and edit it to cover the branch. To see the result of each of the modification to the test case he can click Save and the test case is parsed into EvoSuite test case and executed (covered lines are updated). To facilitate file creation for most of the subjects we created a `File Factory`, where the user can create files to feed test cases.

When the user is finished, he can resume search. When search is resumed, the set of test cases created by the user

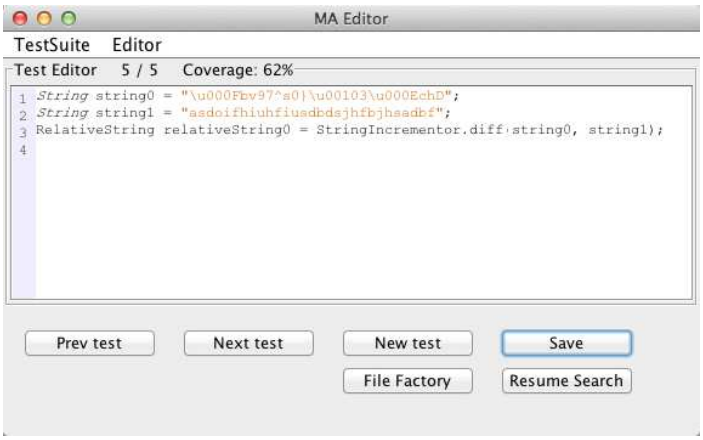

Fig. 5. Test case editor GUI: the editor parses user input, and converts it to EvoSuite's internal format. The user can see the impact of each modification to a test case in Source Code GUI by clicking Save.

are added back to the best individual of the population. To make sure user input is not going to be lost during the genetic algorithm operations, the input is saved and always inserted if needed.

D. Tester

In our context, the test generator provides a summary of potential problems it encounters when generating sequences. There are a number of ways the user can help a test generator for improved effectiveness, structural coverage. For example, the user can write mock objects, object factories, and even sequence rules to accept/reject/underweight/overweight particular test sequences with certain characteristics he or she prefers.

In the screens in Figures 4 and 5, the user can edit test cases that are going to be added to search population inside the best individual. We also added a few mechanisms so user feedback is not going to be eliminated during the genetic algorithm. Also, I/O is a bit of a challenge when it comes to creating test cases. In the editor, the user can use our built-in File Factory to create a set of files that the search can use to create test cases.

IV. Evaluation

This section presents our evaluation of this project.

A. Research Questions

Our experiments can be divided in two groups. The first group contains experiments on a completely controlled environment where it is used automatic test generation only. This first group is used for RQ1 and RQ2 since none of them require user interaction. The second group contains experiments where human input is needed to evaluate our semi-automatic approach and is used to evaluate RQ3 and RQ4. RQ1 and RQ2 were studied as a form of analysis of the EvoSuite and automatic test generation. We wanted to gather data to show the limitations of automatic test generation and to increase the motivation to achieve good results on RQ3 and RQ4. RQ3 and RQ4 both represent the

synergie that the user can have together with an automatic test generation tool in order to improve test efficiency.

RQ1: How often search stagnates and what is the distribution of coverage obtained when it occurs?

RQ2: What is the importance of increasing stagnation bounds?

RQ3: What is the impact on coverage of releasing hard-to-solve test requirements from the search goal?

RQ4: How much can we minimize report to the user?

B. Object of Analysis

We used the SF100 corpus of real Java projects randomly selected from SourceForge [1] as base. This benchmark contains 100 Java projects, including 8784 public classes in total. Fraser and Arcuri provided evidence that this benchmark is valid to evaluate analysis techniques given the diversity of constructions it contains.

C. Setup

In our controlled experiments we have applied `EvoSuite` fully automatic over the whole SF100 benchmark. These experiments are important to answer our primary research questions regarding the behavior of coverage growth over search time. To avoid threat of validity, we also use `EvoSuite`'s built in Sandbox for all SF100 subjects. By using `EvoSuite`'s Sandbox we avoid potentially unsafe operations that can harm the environment and affect next subjects, also every subject has the same runtime conditions.

Since we are working with humans, it seemed intuitive that we understand and study search behavior through time, which is measurable by the user, instead of by number of generations or iterations in the search that can be very variable between different subjects. Therefore, we test search behavior using stagnation bounds based on time. For example, if search coverage doesn't change for 10 seconds, it is considered stagnated, in this case the 10 seconds is the stagnation bound. Notice that stagnation bound is irrespective of the global timeout of the search budget. Conceptually, when the search reaches this bound it means that the search is unable to make progress.

To evaluate this project, we measure human effort to improve test generation results and how much human input improves coverage in contrast to fully automatic approach. We are aware that evaluation using human input can create bias due to user qualification, background, programming skills, etc. and in our experiments we try to avoid that these factors affect the results. We can factor them out by measuring the human effort not just by time needed to report back to the search, but number of lines written by the user, and the number of times the search reached the stagnation bound during out semi-automatic extension execution.

To answer research questions 3 and 4, we selected 20 classes out of SF100. To avoid threats to validity, we chose them based on a testability metric used by Testability Explorer [17], which uses cyclomatic complexity, a software metric commonly used in testability measurement [7], to find hard-to-test classes. This tool computes its testability metric by using recursive cyclomatic complexity based on components inside a method that can't be mocked, it measures the number different paths of execution are there in the code. A class is also penalized in this metric by the number of static fields which are globally reachable by the class under test and could be changed anywhere, global state makes it harder to test. Finally, this tool uses the Law of Diameter [18] to increase the target class score.

Therefore the selected classes will be likely to have branch complexity. We added a few filters to selection, and we limit the number of classes from the same project, such as classes that are mainly Main classes, and also those that show Graphic User Interface. We don't focus in the project GUI testing, and, finally, we select only classes that `EvoSuite` didn't achieve 100% within search budget. The final result of our selection is shown in Table I.

The coverage measurement being used is branch coverage. That means that the test generator goal is to maximize the number of branches that have their predicates satisfied. Even if the commands inside a branch are not fully executed, the branch is counted as covered. It also includes in our goal the execution of methods that don't have any branches.

D. Results

1) Answering RQ1: In order to answer the question of how often search stagnates and its distribution, we conducted this experiment using all subjects of SF100 and fully automated `EvoSuite` and assuming 4 different stagnation bounds, 2, 5, 10 and 15 seconds, and a total of 1 minute time budget (maximum time of exploration). This way we can evaluate the impact of each different stagnation bound. Intuitively, setting this bound too low has the potential risk of reducing coverage and setting it too high has the potential risk of reducing user responsiveness, i.e., search may take more time to respond and time progress will contribute less and less to coverage increase.

The Figure 6 shows the distribution of coverage that `EvoSuite` achieves considering the analysis of each of the classes that reached stagnation bound in separate. We used boxplot notation to represent the distribution. The stagnation was reported in over 50% of the subjects, in particular we can see that the difference of percentage of stagnated classes between stagnation bound of 5 and 15 seconds is very small, more than 90% of stagnations with 5 seconds are still true when increasing the stagnation time in 10 seconds.

Results for the first experiment show that a large number of subjects stagnate in less than 1 minute. It also shows that a small stagnation bound suffices to detect stagnation for most cases of stagnation. We are aware that smaller stagnation bounds can make premature reports more frequently than bigger stagnation bounds, but in real cases it impossible to predict when is the right time to report stagnation.

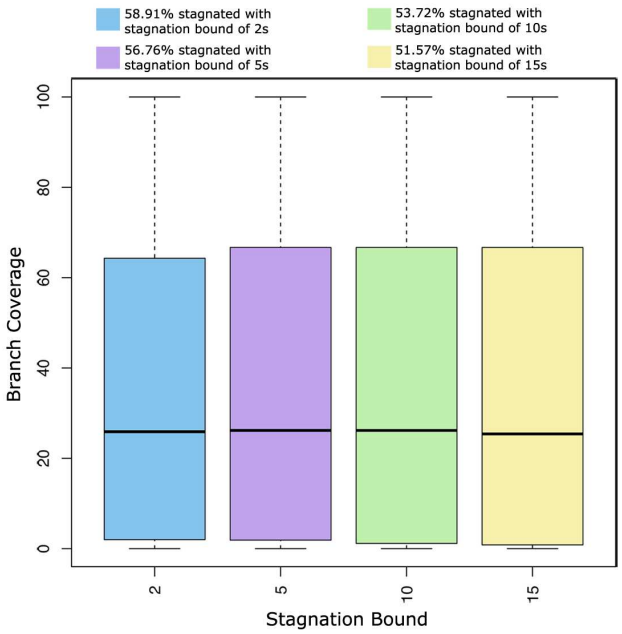

Fig. 6. Coverage distribution during search of subjects that stagnated in each of the stagnation bounds.

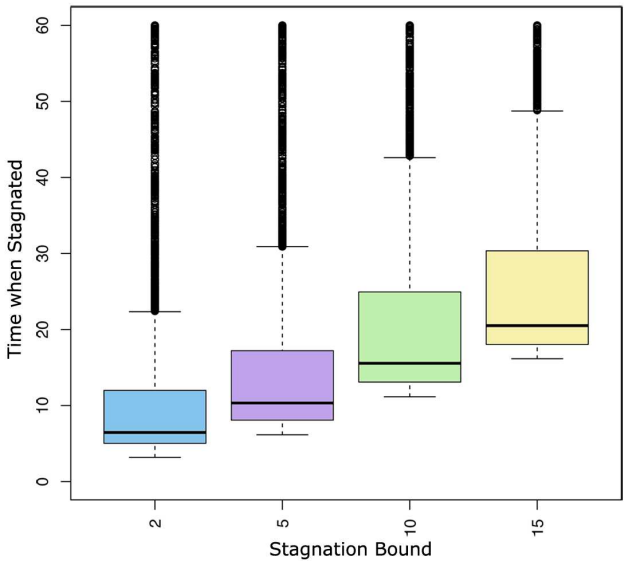

Fig. 7. This boxplot represents the distribution of actual search time that `EvoSuite` stopped in each stagnation bound.

2) Answering RQ2: Given the amount of subjects that stagnated, in this experiment we evaluate this stagnation in matter of global time, how early `EvoSuite` search was considered stagnated for each class that reached stagnation bounds. In this experiment we use the subjects that stagnated in each stagnation bound of the previous experiment and the same search timeout. Clearly, stagnation times with low stagnation bound will be able to be reported to the user sooner than higher stagnation bounds.

The distribution of actual time that search was stopped for each different stagnation bounds is shown in Figure 7 as a boxplot. The behavior described in this plot shows that as soon as the search was able to report stagnation, it did. Most of the stagnation reported happens in the beginning of search time, what makes us believe that the search rapidly found branches in the code that are hard to solve. These spots are important to report to the user, as told before.

The behavior noticed from results of Figures 6 and 7 makes us believe that we might have during search big jumps of coverage. Search increases coverage very fast, then stagnates and when it covers a hard to cover branch, it has another big increase of coverage. This hypothesis is evaluated in the next research question.

3) Answering RQ3: To evaluate the impact of solving a stagnation point, or a hard to cover branch, human input was needed once a stagnation point is reached. We done this experiment ourselves by running our semi-automatic approach. We set a global timeout to search of 2 minutes, this does not count the time that the user needed to give feedback.

The coverage improvement of a given stagnation point is measured by how much it increased until another stagnation or search finalization. For example, if the coverage was 20% at the moment that search stagnated A and user input was requested, and it was 50% in the next stagnation point or the search budget was over, then the coverage improvement for stagnation point A is 30%. So in this experiment we are interested in evaluating the impact of human input by means of coverage improvement of stagnation points in each of the subjects (Table I).

Table II lists the number of stagnation points occurred during the experiment of each subject, and the average of coverage improvement for each of them. It's important to noticed that the whole coverage improvement of a stagnation point is not the user modification of test case. Once new statements ares given back to search (as new individual of population), they are used to solve other similar branches, or mutating into new statements, and by solving branches that are easier to the search but they were dependent on a hard one.

TABLE II. Coverage Improvement by Means of Pauses.

Id	Number of Pauses	Avg. Coverage after Paused
1	3	25%
2	2	7.5%
3	2	17%
4	2	23%
5	2	18%
6	5	19.4%
7	2	41%
8	4	21%
9	2	37.5%
10	2	31%
Balanced Average		**23.30%**

Figure 8 shows a clear case where to advance in code we need to successfully execute the indicated statement. This statement requests an instance of String, but this String must be an address to a file that exists. In Figure 9 is a minimized test case created by `EvoSuite`, the user can simply modify the random String passed to the method to a valid file address or use `FileFactory`, in either way the impact in coverage would be the same. The modified test

TABLE I. SELECTED EXPERIMENT SUBJECTS.

Id	Project	Class	Number of Lines	Number of Goals	Complexity Score
1	Wheel WEB	WheelClassLoader	140	32	99
2	ObjectExplorer	ObjectCopyDialogEventConverter	236	45	99
3	QuickServer	ConfigReader	466	100	197
4	Lilith	LilithBuffer	85	14	202
5	Tullibee	Order	255	22	98
6	celwars2009	Sound	52	11	120
7	at-robots2-j	AtRobotLineLexer	249	119	569
8	TemplateIT	HSSFDataFormat	290	36	239
9	TemplateIT	DynamicTemplate	99	21	267
10	JMCA	JMCAParser	63	11	1628

```
public static void createPatch(String origfn,
      String newfn, String patch, boolean verbose) {
  FileInputStream A = null;
  try {
    if (verbose) {
      System.out.println("Reading file A...");
    }
    String a = StringFromFile.readString(origfn);
                      // Throws IOException
    if (verbose) {
                      // Target branch
      System.out.println("Reading Patch...");
    }
    FileInputStream B = new FileInputStream(patch);
    RelativeString r = new RelativeString(B);
    B.close();
    // ...
  } catch (Exception ex) {
    ex.printStackTrace();
  }
}
```

Fig. 8. This snippet of `FileDiffPatch` class shows one of its methods that `EvoSuite` has problems in covering. Here, `EvoSuite` is not able to reach the indicated branch because the previous statement raises IOException and code flow is redirected to the catch command.

```
String string0 = "Lt <k}hvD}oct\u001F\u0018&1\u001D";
boolean boolean0 = true;
FileDiffPatch.createPatch(string0, string0, string0,
    boolean0);
```

Fig. 9. Best test case created by `EvoSuite` after minimization. It still doesn't cover the target branch described in Figure 8 because it gives a random `String` to `createPatch` instead of a valid file address and `IOException` is raised.

case shown in Figure 10 immediately increases coverage of the subject from 36.84% to 44.44% and after new generation are created, the coverages improves to 100%.

Since search time was limited first by time then coverage (obviously, search is finished once it reaches 100%),

```
String string0 = "Lt <k}hvD}oct\u001F\u0018&1\u001D";
boolean boolean0 = true;
File file0 = FileFactory.getRandomFile();
String string1 = file0.getAbsolutePath();
FileDiffPatch.createPatch(string1, string0, string0,
    boolean0);
```

Fig. 10. Test case from Figure 9 after manual editing to create and pass a valid file address to createPatch.

TABLE III. SUMMARY OF BRANCH COVERAGE

Id	EvoSuite	Semi-Automated	Infeasible Branches
1	28.13%	100%	2
2	31.11%	46%	10
3	13.00%	100%	17
4	57.14%	100%	8
5	3.7%	100%	0
6	18.18%	100%	23
7	17.65%	100%	0
8	27.78%	100%	3
9	42.86%	100%	1
10	27.27%	100%	0

some of the resulting coverages didn't reach 100%. Table III summarizes the obtained coverages by fully automated `EvoSuite` and semi-automated approach. Also, the infeasible branches that were reported had their percentage in code were considered as covered, it includes their dependent branches.

In our experiments, semi-automatic test generation increased branch coverage by by 67.9%. Such impressive improvement is not only caused by the user input, but the set of low testability classes that were selected with Testability Explorer. Also, the number of infeasible branches that were found is a variable in this metric. With no report of infeasible branches, search would go on until its budge ended trying to move on, but it would impossible.

4) Answering RQ4: This experiment evaluates how much minimization can be done on the best test cases that we found for the frontier branches (next to cover branches) when a stagnation bound is reached. Frontier branches are those branches that were not covered yet but their dependencies have been, so they are the next branches to be covered. During search, we keep track of the test cases that got the closest (smaller distance) to each frontier branch, once a stagnation bound is reached, and the search is stopped to give the user feedback, these test cases are minimized to keep the same distance to their goal and have the minimum size as possible.

Figure 11 shows how effective the minimization of test cases is. Though `EvoSuite`'s search focuses on also minimizing its test cases, when search stagnates it might not have enough time to have minimized them. Before minimization the longest test had over 93 lines, and after minimization the longest has 43 lines. We may still have a lot of work left to never show the user a test case as big as 43 lines.

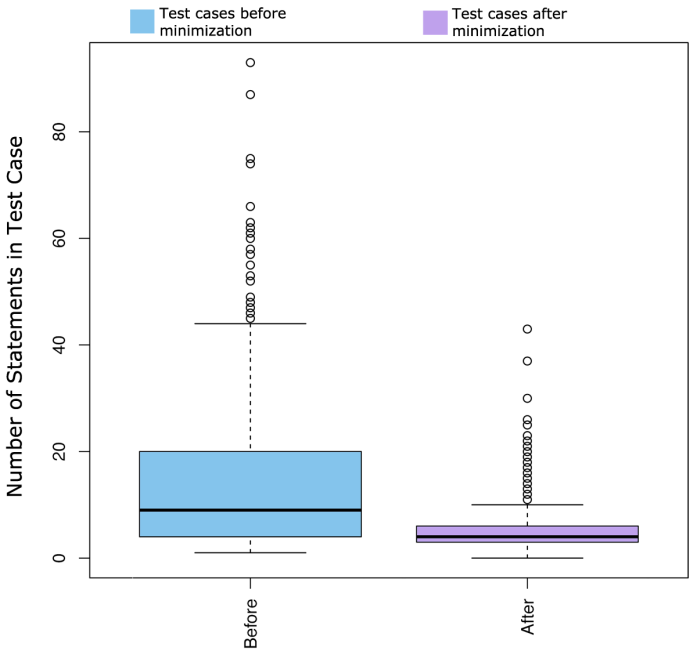

Fig. 11. Boxplot distribution of test case length before and after minimization.

V. Conclusions and Future Work

In this paper, we have presented a semi-automatic approach and had good results that show that a minor human input can help improve test generator coverage. We implemented this approach as an extension of `EvoSuite` to get dynamic data from search and give back to it feedback from the user to cover new branches. Our experiments to understand search behavior showed us that it is important for performance that human input release search from hard-to-cover areas.

There is a real potential in including human in the loop of test generation, and a lot can still be done in the areas to communicate with the user: simplify information as most as possible to be able to not even need source code nor test cases, so it would be possible for companies to use crowd-sourcing without the need to jeopardize their security.

References

[1] G. Fraser and A. Arcuri, "Sound empirical evidence in software testing," in *ICSE*, 2012, pp. 178–188.

[2] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux, "Precise identification of problems for structural test generation," in *ICSE*, 2011, pp. 611–620.

[3] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *ESEC/FSE*, 2011, pp. 416–419.

[4] *Randoop website*, http://code.google.com/p/randoop/.

[5] *Pex and Fakes (Moles) website*, http://research.microsoft.com/en-us/projects/pex/.

[6] K. Sen and G. Agha, "CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools," in *CAV*, 2006, pp. 419–423.

[7] M. Bruntink and A. v. Deursen, "Predicting class testability using object-oriented metrics," in *Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, ser. SCAM '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 136–145. [Online]. Available: http://dx.doi.org/10.1109/SCAM.2004.15

[8] B. Daniel and M. Boshernitsan, "Predicting effectiveness of automatic testing tools," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 363–366. [Online]. Available: http://dx.doi.org/10.1109/ASE.2008.49

[9] J. H. Andrews, A. Groce, M. Weston, and R.-G. Xu, "Random test run length and effectiveness," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 19–28. [Online]. Available: http://dx.doi.org/10.1109/ASE.2008.12

[10] G. Fraser and A. Arcuri, "The seed is strong: Seeding strategies in search-based software testing," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ser. ICST '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 121–130. [Online]. Available: http://dx.doi.org/10.1109/ICST.2012.92

[11] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 78–88. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336763

[12] Y. Pavlov and G. Fraser, "Semi-automatic search-based test generation," in *5th International Workshop on Search-Based Software Testing (SBST'12) at ICST'12*, 2012, pp. 777–784.

[13] H.-S. Kim and S.-B. Cho, "Application of interactive genetic algorithm to fashion design," in *Engineering Applications of Artificial Intelligence*, 2000, p. 635âĂŞ644.

[14] N. Chen and S. Kim, "Puzzle-based automatic testing: bringing humans into the loop by solving puzzles," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 140–149. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351697

[15] F. Pastore, L. Mariani, and G. Fraser, "Crowdoracles: Can the crowd solve the oracle problem?" in *ICST'13: Proceedings of the 6th International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 2013, to appear.

[16] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *Proceedings of the 2011 11th International Conference on Quality Software*, ser. QSIC '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 31–40. [Online]. Available: http://dx.doi.org/10.1109/QSIC.2011.19

[17] M. Hevery, "Testability explorer: using byte-code analysis to engineer lasting social changes in an organization's software development process." in *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, ser. OOPSLA Companion '08. New York, NY, USA: ACM, 2008, pp. 747–748. [Online]. Available: http://doi.acm.org/10.1145/1449814.1449842

[18] *Law of Demeter - wikipedia*, http://en.wikipedia.org/wiki/Law_of_Demeter.