JFIX: Semantics-Based Repair of Java Programs via Symbolic PathFinder

Xuan-Bach D. Le
Singapore Management University
Singapore
dxb.le.2013@smu.edu.sg

Duc-Hiep Chu
Institute of Science and Technology
Austria
duc-hiep.chu@ist.ac.at

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

Claire Le Goues
Carnegie Mellon University
Pittsburgh, USA
clegoues@cs.cmu.edu

Willem Visser
Stellenbosch University
South Africa
wvisser@cs.sun.ac.za

ABSTRACT

Recently there has been a proliferation of automated program repair (APR) techniques, targeting various programming languages. Such techniques can be generally classified into two families: syntactic- and semantics-based. Semantics-based APR, on which we focus, typically uses symbolic execution to infer semantic constraints and then program synthesis to construct repairs conforming to them. While syntactic-based APR techniques have been shown successful on bugs in real-world programs written in both C and Java, semantics-based APR techniques mostly target C programs. This leaves empirical comparisons of the APR families not fully explored, and developers without a Java-based semantics APR technique. We present JFix, a semantics-based APR framework that targets Java, and an associated Eclipse plugin. JFix is implemented atop Symbolic PathFinder, a well-known symbolic execution engine for Java programs. It extends one particular APR technique (Angelix), and is designed to be sufficiently generic to support a variety of such techniques. We demonstrate that semantics-based APR can indeed efficiently and effectively repair a variety of classes of bugs in large real-world Java programs. This supports our claim that the framework can both support developers seeking semantics-based repair of bugs in Java programs, as well as enable larger scale empirical studies comparing syntactic- and semantics-based APR targeting Java. The demonstration of our tool is available via the project website at: https://xuanbachle.github.io/semanticsrepair/

CCS CONCEPTS

• **Software and its engineering → Error handling and recovery**; *Software testing and debugging*;

KEYWORDS

Automatic Program Repair, Symbolic Execution, Program Synthesis

ISSTA'17, July 10–14, 2017, Santa Barbara, CA, USA

ACM ISBN 978-1-4503-5076-1/17/07…$15.00
https://doi.org/10.1145/3092703.3098225

ACM Reference format:
Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. JFIX: Semantics-Based Repair of Java Programs via Symbolic PathFinder. In *Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10–14, 2017 (ISSTA'17),* 4 pages.
https://doi.org/10.1145/3092703.3098225

1 INTRODUCTION

Bug fixing is known to be difficult, time-consuming, laborious, and expensive. Automated program repair (APR) techniques that can help developers tackle the bug-fixing challenge would thus be of tremendous value. Numerous recent research advances have brought the once-futuristic idea of APR closer to reality [8, 9, 11, 13–15, 17]. APR techniques can be generally categorized as two families: syntactic- (or heuristic) versus semantics-based. Syntactic APR techniques typically generate a large number of possible candidate bug-fixing patches by manipulating or applying repair templates to the abstract syntax tree of buggy programs. Recent works in this family include PAR [6], and HDRepair [11] that can repair many large, real-world Java programs. Meanwhile, semantics-based techniques typically use symblic reasoning to infer semantics constraints, or *specifications* from behavior on test suites, leveraging program synthesis to synthesize repaired expressions that conform to the extracted specifications. A recent semantics-based APR tool named Angelix [17] has demonstrated good scalability in repairing bugs in large real-world programs written in C, comparable to those targeted by syntactic approaches. Recent APR techniques target different programming languages, namely C and Java; this can make it difficult to empirically compare all such tools. Also, although Java is the most popular programming language and its influence is growing over time,[1] yet there are only a few successful repair tools targeting Java [4, 6, 11], with some C-based tools translated accordingly (such as in the Astor framework [16]). Only a few of them have publicly available implementations [4, 11]. With the exception of Nopol [4], semantics-based techniques target and are implemented for C programs.

We demonstrate JFix, a repair framework and an associated Eclipse plugin that bring the strength of semantics-based APR approach to repair Java programs. We have two primary users in mind that inform our framework design: (1) *Researchers*, for whom

[1] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

we seek to provide an extensible semantics-based repair framework that targets Java, enabling extensions of and empirical comparisons with Java-focused APR techniques, and (2) *Developers*, for whom we seek to provide real-world semantics-based Java repair facilities. We therefore translate and extend Angelix [17], a traditional semantics-based APR technique, such that it can target Java, making use of a selective symbolic execution procedure based on Symbolic PathFinder [19]. We further implement several additional features in support of our primary goals:

- **Extensibility.** We implement repair specification inference in a *generic* fashion, to support a framework that can integrate many different synthesis engines [10]. Our experiments show that our framework can repair more bugs by using multiple synthesis engines.
- **Java-specificity.** We extend our tool to infer specifications that could help repair bugs related to *method calls*, that traditional semantics-based APR such as Angelix is not yet able to handle.[2] Bugs related to method calls are prevalent in Java programs [18], and thus, JFix's ability to repair these bugs would be valuable.
- **Usability.** Angelix requires users to manually instrument the program under repair for the specification inference task, that could be tedious and error-prone. As opposed to Angelix, we alleviate the burden of manually inspecting and instrumenting the program under repair (discussed in Section 2.1). We also provide an Eclipse plugin associated with JFix to bring it to wider classes of users.

We demonstrate JFix on a small set of 47 Java bugs from the IntroClass benchmark [12], and nine real bugs from large real-world software. The results show that JFix's unique ability in leveraging multiple synthesis engines allows it to repair more bugs as compared to using a single synthesis engine alone. JFix can repair bugs with various types, including those involving method calls, and at scale. Also, JFix is able to generate multi-line fixes. This promising result suggests that JFix can enable larger empirical comparison of repair tools targetting Java programs in the future, and facilitate an open research environment. A demonstration of JFix is available at https://xuanbachle.github.io/semanticsrepair/.

The remainder of this paper is as follows: Section 2 explains JFix. Experimental results are presented in Section 3, followed by discussion, conclusion and future work in Section 4.

2 SEMANTICS-BASED REPAIR WITH JFIX

Figure 1 depicts the flowchart of our framework. Assume as input a Java program and a set of JUnit test cases, at least one of which is failing; the goal is to modify the program such that all test cases pass. Given the input, the JFix front-end first instruments the program, and runs tests to collect traces. These traces are then input to the fault localization module to identify likely-buggy locations. This module outputs top-ranked locations, which are later given to the JFix back-end. The back-end performs selective symbolic execution to obtain error-free execution paths and path conditions, which can then be used to infer the specifications. The inferred specifications inform the synthesis of replacement code (patch) for the buggy expressions. If the patched program does not pass validation tests, JFix continues the process with newly chosen buggy locations, until a patch that passes all validation tests is found (or the set of candidate buggy locations is exhausted).

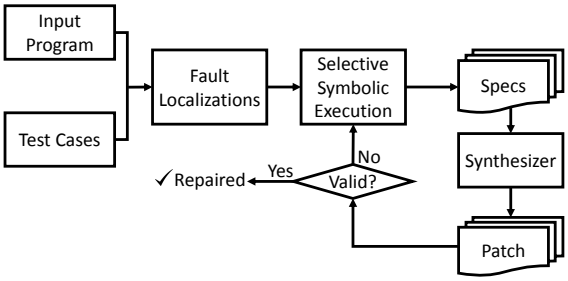

Figure 1: Flowchart of JFix's approach.

Next, we detail JFix's approach. We then describe an Eclipse plugin for the tool.

2.1 Overall Approach

We illustrate the approach via the example in Figure 2, a real bug from the Commons Math library [1] at revision *e5c7e40*: the parameter *maxUlps* is used instead of *epsilon* in a method call, which is wrapped inside a loop.

```
1   for (int i = getNumObjectiveFunctions(); i <
          getArtificialVariableOffset(); i++) {
2       final double entry = tableau.getEntry(0, i);
3   -   if (Precision.compareTo(entry, 0d, maxUlps) > 0)
4   +   if (Precision.compareTo(entry, 0d, epsilon) > 0)
5       {
6        columnsToDrop.add(i);
7       }
8   }
```

Figure 2: A bug in Commons Math library, revision *e5c7e40*. The bug is at line 3, and the corresponding developer-submitted fix is depicted at line 4.

First, JFix uses Ochiai [2], a fault localization metric, to identify likely-buggy locations; in our example, it identifies line 3. JFix then installs symbolic variable(s), α_i, to represent the selected location(s).[3] Particularly, the code at line 3 in our example would become `if(`α`)`. JFix runs symbolic execution on the instrumented program against test suites to collect path conditions that do not lead to any test failures. By this way, JFix only symbolically reasons about certain *selected* likely-buggy program locations, allowing it to scale to large programs (demonstrated in Section 3). We implemented this symbolic execution atop Symbolic PathFinder [19].

Solving the failure-free path conditions returns constraints over α that lead the tests to pass; These constraints serve as a postcondition on the code at α. The precondition is represented by the collection of runtime values of visible variables, fields, and method calls at the buggy expression(s) before executing the expression(s), e.g., variables named $i, entry, maxUlps, epsilon$, and method calls such as *Precision.compareTo(entry, 0d, epsilon)* in our example. These specifications are then passed to a synthesis engine to construct possible replacement expressions, consistent with prior work [10, 17].

Although the specification inference task in JFix shares the same spirit with Angelix [17], there are key differences. Angelix requires

[2] JFix can fix bugs in programs that are written in Java 7 and 8, and not related to lambda functionality.

[3] For simplicity, we describe the process with respect to a single-line fix. However, the approach generalizes to multi-line fixes by installing multiple symbolic variables.

users to manually instrument the program under repair in order to infer specifications: Users need to identify relevant output variables of the program and specify desired outputs for them.[4] This can be an especially daunting and error-prone task when the program under repair is large and involves many outputs (e.g., an array of integers).

JFix automatically infers specifications by leveraging the capabilities of Java PathFinder (JPF) [22]. JPF appropriately interprets assertions, a necessary part of JUnit test cases, in an automated manner. JFix thus inherits this functionality from JPF since it is implemented atop Symbolic PathFinder [19]. This allows JFix to automatically separate error-free execution paths from the buggy ones. Subsequently, JFix can automatically extract the specifications without requiring manual instrumentation from users.

We next discuss runtime values collection schemes in Section 2.2, that support the specification inference phase.

2.2 Runtime Values Collection

We allow users to instruct JFix to *selectively* collect runtime values that are of potential interest. This enables JFix to filter out potentially unnecessary/irrelevant runtime values, subsequently making the inferred specifications more concise.

We collect runtime values of various sources capturing common bug fixing scenarios [18]. The sources are described bellow; note that "in-scope" is always used with respect to the considered buggy line(s).

- Values of fields of the buggy class.
- Values of in-scope local variables.
- Values of variables used elsewhere in the same file.
- Values of variables (locals, fields) used at the buggy lines.
- Result of method calls used elsewhere in the same file. The collected calls should be return type-compatible with those used at the buggy lines. This allows method call substitution, such as shown in Figure 2.
- Result of method calls used in the buggy lines, but with one or more different parameters.
- Result of methods calls declared in the same class that are compatible (e.g., same return type) with those used at the buggy lines.

We allow users to specify (a combination of) only a small number of sources, for which the runtime values will be collected.

2.3 Multiple Synthesis Solvers

Once the specifications are inferred, synthesis solvers will be used to synthesize repairs conforming to the specifications. An interesting feature of JFix is the ability to use multiple synthesis solvers as opposed to traditional semantics-based repair approach such as Angelix [17] which only uses a single synthesis solver alone. JFix implements a procedure that translates the inferred specifications into a syntax guided synthesis (SyGuS) grammar, allowing it to employ the SyGuS solvers [10].[5] Particularly, JFix can use the synthesis engine of Angelix [17], and state-of-the-art SyGuS solvers namely Enumerative and CVC4 [3].

[4]See Angelix's tutorial for more details:
https://github.com/mechtaev/angelix/blob/master/doc/Tutorial.md
[5]We refer interested reader to the detailed translation described in [10]

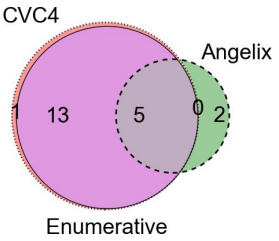

Figure 3: Non-overfiting repairs generated by JFix using multiple synthesizers, namely Angelix, Enumerative, and CVC4

2.4 Eclipse Plugin

We also implemented an Eclipse plugin that supports a user interface with JFix. The developer can configure our plugin in Eclipse programming environment with various options (such as the runtime collection schemes described in previous section 2.2), to repair the buggy program. The demonstration video of JFix and its Eclipse plugin is available at: https://xuanbachle.github.io/semanticsrepair/.

3 EMPIRICAL EVALUATION

We show that JFix's unique ability in employing multiple synthesis engines allows it to repair more bugs relative to Angelix which is a traditional semantics-based repair that uses a single synthesis engine alone [17].

RQ1. How frequently can JFix generate generalizable repairs? Can JFix generate multi-line repairs?

We evaluated JFix on 47 bugs in the *smallest* subject program from a Java implementation of the IntroClass benchmark consisting of student written homework assignments in a freshmen class [12]. The *smallest* subject program finds the smallest number among four integer numbers. Each program version is accompanied by two test suites: black-box tests manually written by the class instructor, and white-box tests automatically generated by the automatic test generation tool. Note that we use the Java versions of the dataset available in [5].

We use black-box tests to generate repair. White-box tests are later used as *held-out* tests to assess if generated repairs overfit to the test cases used for repair. This validation method is a proxy for an *objective* assessment of repair quality that is often used in prior works, e.g., [10, 20]. If a repair does not pass the white-box tests, we say the repair does not generalize, or overfits to the training set, and is thus of lower quality than likely desired. Repair techniques that create more generalizable patches are better on the whole.

Figure 3 shows the number of non-overfitting/generalizable repairs, which pass all tests including the held-out tests, generated by the synthesis engines, namely Angelix 's synthesis engine (denoted as Angelix), Enumerative, and CVC4. The result shows that JFix's unique ability in employing multiple synthesis engines can enhance the capability of generating more non-overfitting repairs. For example, there are 14 bugs that Enumerative and CVC4 can generate non-overfitting repairs while Angelix's synthesis engine cannot generate.

JFix is also able to generate multi-line patches. Figure 4 shows a multi-line patch generated by JFix using Angelix's synthesis engine for a *smallest* program version. The patch involves simultaneous changes at lines 3, 8, and 13.

RQ2. Can JFix scale to large, real-world software?

We applied JFix to nine real bugs from large real-world software including Commons Math library [1], which contains 175 kLOC in the package.

JFix successfully repairs all bugs by using multiple synthesis solvers. To assess the correctness of each generated repair, we check whether the repair is equivalent to the repair submitted by developers. A repair is equivalent to the developer-submitted fix (indicated by ✔ in Table 1) if it can be transformed to the developer's fix by basic syntactic transformations (as similarly conducted in [?]). For example, `(a || b)` and `(b || a)` are considered equivalent by swapping left and right hand sides of the operator `||`. We also note that JFix can efficiently repairs bugs in up to 4 minutes. In summary, the results show that JFix is scalable and efficient.

Table 1: Real-world software bugs fixed by JFix. "Rev" shows bug-fix revisions. "Type" shows bug types: "I" denotes method call, "II" denotes arithmetic. "Time" shows repair time (in seconds) ("NA" denotes not available). "Dev" indicates developer-equivalent patch: "✔" denotes equivalent, and "✗" denotes otherwise.

Project	Rev	Type	Angelix		Enum		CVC4	
			Time	Dev	Time	Dev	Time	Dev
Math	09fe	II	23s	✔	26s	✔	36s	✔
	ed5a	I & II	168s	✔	NA	✗	NA	✗
Jflex	2e82	II	NA	✗	70s	✔	72s	✔
Fyodor	2e82	II	20s	✔	19s	✔	31s	✔
SFM	5494	II	12s	✗	10s	✔	13s	✔
EWS	299a	I	NA	✗	14s	✔	258s	✔
Orientdb	b33c	II	20s	✔	22s	✔	NA	✗
Qabel	299c	II	37s	✔	22s	✗	23s	✗
Kraken	8b0f	II	12s	✔	13s	✗	15s	✗

```
1       if (a.value > b.value) {
2           m.value = b.value;
3       } else if (a.value ≤ b.value) {//changed < to ≤
4           m.value = a.value;
5       } if (m.value > c.value) {
6           n.value = c.value;
7       } else if (m.value ≤ c.value) {//changed < to ≤
8           n.value = m.value;
9       } if (n.value > d.value) {
10          p.value = d.value;
11      } else if (n.value ≤ d.value) {//changed < to ≤
12          p.value = n.value;
13      }
14      println("smallest number is: \%d", p.value)
```

Figure 4: A multi-line fix by JFix for a *smallest* program version. The goal of the program is finding the smallest number among the values of *a*, *b*, *c*, and *d*.

4 DISCUSSION & CONCLUSION

JFix's approach is sound in the sense that the repairs generated by JFix pass all tests. We further assess the quality of generated repairs by various proxies: (1) use an independent test suite for testing the repairs, (2) check whether the repairs are equivalent to the fixes submitted by developers. Regarding bugs that JFix currently cannot repair, we found that these bugs often require adding new code structure, e.g., adding a branch `else{...}` to fix. This is actually a common drawback of semantics-based repair systems in general [17].

In summary,we presented JFix – a Java-targeted semantics-based repair framework that consists of several repair synthesis engines, and its associated Eclipse plugin. JFix can repair various bug types, including bugs that involve multiline changes or method calls, and scale to large real-world programs. As future work, we plan to enhance JFix by employing specification mining tools (e.g., [7]) to strengthen the inferred specifications, and automatically detecting defect types (c.f., [21]) as a step to customize repair techniques to fix specific defects more efficiently.

ACKNOWLEDGMENTS

We thank Vu Le (Microsoft Research, Redmond), and anonymous reviewers for their comments. Duc-Hiep Chu was supported in part by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE/SHiNE) and Z211-N23 (Wittgenstein Award).

REFERENCES

[1] 2016. Apache common Math library. (2016). http://commons.apache.org/proper/commons-math/
[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *TAICPART-MUTATION 2007*.
[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2015. Syntax-guided synthesis. *Dependable Software Systems Engineering* (2015).
[4] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic repair of buggy if conditions and missing preconditions with smt.
[5] Thomas Durieux and Martin Monperrus. 2016. *IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs.* Technical Report.
[6] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *ICSE*.
[7] Tien-Duy B Le, Xuan-Bach D Le, David Lo, and Ivan Beschastnikh. 2015. Synergizing specification miners through model fissions and fusions (t). In *ASE*.
[8] Xuan Bach D. Le, Quang Loc Le, David Lo, and Claire Le Goues. 2016. Enhancing Automated Program Repair with Deductive Verification. In *ICSME*.
[9] Xuan-Bach D Le, Tien-Duy B Le, and David Lo. 2015. Should fixing these failures be delegated to automated program repair?. In *ISSRE*.
[10] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. Empirical study on synthesis engines for semantics-based program repair. *ICSME*.
[11] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *23rd SANER*.
[12] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE TSE* (2015).
[13] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *TSE* (2012).
[14] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *ESEC/FSE*.
[15] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *ACM SIGPLAN Notices*.
[16] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java (Demo) *(ISSTA 2016)*. 441–444.
[17] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *ICSE*.
[18] Kai Pan, Sunghun Kim, and E. James Whitehead, Jr. 2009. Toward an Understanding of Bug Fix Patterns. *Empirical Software Engineering* (2009).
[19] Corina S Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *ASE journal* (2013).
[20] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *ESEC/FSE*.
[21] Ferdian Thung, Xuan-Bach D Le, and David Lo. 2015. Active semi-supervised defect categorization. In *ICPC*.
[22] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. 2000. Model Checking Programs. *Automated Software Engineering* (2000).