Contract-Based Program Repair without The Contracts: An Extended Study

Liushan Chen, Yu Pei, Carlo A. Furia

Abstract—Most techniques for automated program repair (APR) use tests to drive the repair process; this makes them prone to generating spurious repairs that overfit the available tests unless additional information about expected program behavior is available. Our previous work on JAID, an APR technique for Java programs, showed that constructing detailed state abstractions—similar to those employed by techniques for programs with contracts—from plain Java code without any special annotations provides valuable additional information, and hence helps mitigate the overfitting problem.

This paper extends the work on JAID with a comprehensive experimental evaluation involving 693 bugs in three different benchmark suites. The evaluation shows, among other things, that: 1) JAID is effective: it produced correct fixes for over 15% of all bugs, with a precision of nearly 60%; 2) JAID is reasonably efficient: on average, it took less than 30 minutes to output a correct fix; 3) JAID is competitive with the state of the art, as it fixed more bugs than any other technique, and 11 bugs that no other tool can fix; 4) JAID is robust: its heuristics are complementary and their effectiveness does not depend on the fine-tuning of parameters. The experimental results also indicate the main trade-offs involved in designing an APR technique based on tests, as well as possible directions for further progress in this line of work.

1 Introduction

Every general software analysis technique based on a finite collection of tests is prone to *overfitting* them. In automated program repair (APR), overfitting is likely to cripple the performance of tools following the popular *generate-then-validate* paradigm [1]–[3], since passing validation against a finite—often small—number of tests does not guarantee that a repair is genuinely *correct* from a programmer's perspective.

Leveraging additional information about a program's intended behavior is key to mitigating the risk of overfitting. The AutoFix technique for APR [4] used *contracts*, made of *assertions* such as pre- and postconditions, as a source of additional information; but support for contracts is limited or non-existent in the most widely-used programming languages. Nonetheless, our previous work [5] showed that it is still possible to generalize some of the techniques for automated program repair based on contracts so that they work on plain object-oriented code (without any contracts). To this end, we introduced JAID: a technique and tool for *automated program repair* of Java programs is based on detailed, *state-based* dynamic program analyses, which can extract information from regular code that helps construct high-quality fixes. JAID was the first generate-then-validate APR technique that achieved high levels of precision *without relying on additional input* other than tests and faulty code; in contrast, other high-precision APR techniques following the same paradigm [6], [7] analyze a large number of project repositories to collect additional information that guides fixing.

• *Liushan Chen and Yu Pei are with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China.*
E-mail: {cslschen,csypei}@comp.polyu.edu.hk.
• *Carlo A. Furia is with the Software Institute of USI Università della Svizzera italiana, Lugano, Switzerland.*
Homepage: https://bugcounting.net/.

Received: 2019-05-12; revised: 2019-12-09; accepted: 2020-01-15.

The present paper extends the previous work on JAID [5] with a thorough experimental evaluation of its capabilities based on a revised implementation of the tool. The key novel **contributions** include:

- an experimental evaluation of JAID on 693 bugs from three different benchmark suites;
- a detailed assessment of JAID's effectiveness (how many bugs it can fix), efficiency (how quickly it runs), and used heuristics (for fault localization, fix generation, and ranking);
- a quantitative comparison of JAID with all automated program repair tools for Java that are available at the time of writing;
- a revised implementation of JAID, which remains available as open source.

JAID's revised implementation does not change the technique's fundamentals and its overall design (which we summarize in Sec. 2) but improves several details of the tool that collectively make it more widely applicable: 1) JAID now supports some exception-related Java language behavior that were not handled correctly in earlier versions; 2) it avoids crashes in the program state monitoring stage that led to blocking or failed expression evaluation in earlier versions; 3) it performs validation more efficiently, by detecting duplicate (identical) fixes. Sec. 2.7 provides details about these improvements to JAID's implementation.

These changes helped extend the programs JAID can actually handle, as demonstrated in the extensive experimental evaluation of Sec. 3 whose **results** include:

- JAID produced correct fixes for over 15% of the bugs;
- on average, JAID took less than 30 minutes to produce a correct fix;
- JAID could correctly fix more bugs than any other APR

tool for Java at the time of writing, including 11 bugs that no other tool can fix;
- JAID's heuristics are effective and fairly robust (that is, JAID's behavior does not depend on fine-tuning its parameters).

These results indicate that JAID is a competitive tool at the time of writing, achieving a combination of effectiveness, efficiency, and applicability that often compares favorably to the state of the art. They also outline directions to further improve the overall performance of JAID's algorithm: improve the precision of fault localization and fix ranking, and reduce analysis times.

Outline. Reflecting the focus on empirical analysis, the bulk of the paper describes the design of the experimental evaluation (Sec. 3), its detailed results (Sec. 4), and threats to validity (Sec. 5). To make the paper self contained, Sec. 2 gives a high-level view of JAID's techniques; the conference paper [5] provides more details about how JAID works.

Terminology. In this paper we use the nouns "defect", "bug", "fault", and "error" as synonyms to indicate errors in a program's source code; and the nouns "fix", "patch", and "repair" as synonyms to indicate source-code modifications that ought to correct errors. For simplicity, JAID denotes both the APR technique and the tool implementing it.

Availability. JAID and all the material of the experiments described in this paper is available as open source at:
https://bitbucket.org/maxpei/jaid

2 How Jaid Works

JAID follows the popular "generate-then-validate" approach, which first generates a number of candidate fixes, and then validates them using the available test cases; Fig. 1 gives an overview of the overall process. Inputs to JAID are a Java program, consisting of a collection of classes, and test cases that exercise the program and expose some failures. Since JAID's approach has not changed but in minor implementation details since its conference publication [5], this section makes the present paper self-contained but does not introduce major novel concepts or techniques.

One key feature of JAID is how it abstracts and monitors program state in terms of program expressions; all stages of JAID's workflow rely on the abstraction derived as described in Sec. 2.1. The rest of this section gives some details of how JAID works; the presentation targets the repair of a generic method `fixMe` of class `FC`, with tests T that exercise `fixMe` in a way that at least one test in T is failing.

2.1 Program State Abstraction

JAID bases its program analysis and fix generation processes on a detailed *state-based abstraction* of the behavior of method `fixMe`. For every location ℓ in `fixMe`, uniquely identifying a statement in the source code, JAID records the values of a set M_ℓ of expressions during each test execution: 1) the exact value of expressions of numeric and Boolean types; 2) the object identifier (or **`null`**) of expressions of reference types, so that it can detect when a reference is aliased, or is **`null`**.

JAID builds the set M_ℓ of *expressions* so that it includes: 1) all basic expressions from `fixMe` (variables and complete

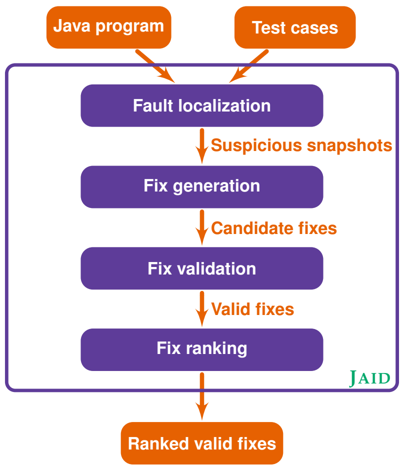


Fig. 1: An overview of how JAID works. Given a Java program and a set of test cases, including at least one failing test, JAID identifies a number of suspicious snapshots, each indicating a location and an abstraction of the program state at that location that may be implicated in the failure; based on the snapshot information, JAID generates a number of candidate fixes, which undergo validation against all available tests for the method under repair; fixes that pass all available tests are considered valid; JAID finally heuristically ranks the valid fixes, and presents the valid fixes to the user in ranking order.

expressions used in the method body) that are visible at ℓ; and 2) all extended expressions obtained by performing argumentless calls on basic expressions of reference type. While monitoring the value of expressions in M_ℓ, JAID identifies which are *pure*—that is, their evaluation does not obviously change the program state. Only expressions that are pure according to this dynamic analysis are retained in M_ℓ and used for state abstraction and monitoring.

2.2 Fault Localization

The goal of fault localization is to identify *suspicious snapshots* indicating locations and states that are likely to be implicated in a fault. A *snapshot* is a triple $\langle \ell, b, ? \rangle$, where ℓ is a location in method `fixMe` under repair, b is a Boolean expression, and $?$ is the value (**`true`** or **`false`**) of b at ℓ. JAID includes snapshots for Boolean expressions that can be built from expressions in M_ℓ as well as comparison and logical operators like ==, <=, and &&.

Snapshot suspiciousness. The *suspiciousness* value $susp(s)$ of a snapshot $s = \langle \ell, b, ? \rangle$ combines two sources of information: 1) a syntactic analysis of expression dependence, which gives a higher value ed_s to s the more subexpressions b shares with those used in the statements immediately before and immediately after ℓ (this estimates how much s is relevant to capture the state change at ℓ); 2) a dynamic analysis, which gives a higher value dy_s to s the more often b evaluates to $?$ at ℓ in a failing test, and a lower value to s the more often b evaluates to $?$ at ℓ in a passing test (this collects the evidence that comes from monitoring the program during passing and failing tests). The overall suspiciousness $susp(s) = 2/(ed_s^{-1} + dy_s^{-1})$ is the harmonic mean of these two sources, but the dynamic analysis has a greater impact—because ed_s is set up to be a value between zero and one, whereas dy_s is at least one and grows with the number of passing tests.

Since this proved to work well in practice, JAID's fault localization algorithm follows Heuristic III from Wong et al.'s approach [8], and applies it to the localization of snapshots. However, we have evidence that JAID's overall effectiveness is largely independent of the details of how fault localization is done: Sec. 4.5 discusses the results of replacing JAID's fault localization algorithm with five other "off-the-shelf" techniques, showing that the number of bugs that JAID can fix barely changes.

2.3 Fix Generation: Fix Actions

A snapshot $s = \langle \ell, b, \texttt{?} \rangle$ with high suspiciousness indicates that the program is prone to triggering a failure when the *program state* in some execution is such that b evaluates to `?` at ℓ; correspondingly, JAID builds a number of candidate fixes that try to *steer away* from the suspicious state in the hope of avoiding the failure. To this effect, JAID enumerates three kinds of *fix actions*:

Semantic actions are based on JAID's rich state-based abstractions, and build patches that modify the state (directly or indirectly) using the expressions captured by snapshots and ranked by dynamic analysis; they include assignments (which mutate the state directly) and modifications of existing expressions (which affect the state indirectly).

Mutation actions are simple, "syntactic" mutations of an existing statement that capture common sources of programming mistakes such as off-by-one errors.

Control-flow actions modify the program's control flow by changing a branching condition or by adding an abrupt termination statement (**`break`** or **`return`**).

By using the information of suspicious snapshots, JAID can generate many *different concrete fix actions* in the same category. This gives JAID the flexibility needed to be able to fix a variety of different bugs; Sec. 4.4 describes which actions turned out to be most useful in the experiments. Following are the details of how JAID builds fix actions.

Derived expression. Given an expression e, $\Delta_{\ell,e}$ denotes all derived expressions built from e as follows: 1) if e has integer type, $\Delta_{\ell,e}$ includes e, e `+ 1`, and e `- 1`; 2) if e has Boolean type, $\Delta_{\ell,e}$ includes e and `!`e; 3) $\Delta_{\ell,e}$ also includes t and t`.f(`$\cdots$`)`, for every $t \in M_\ell$ of reference type, where `f` is a function of the class t belongs to—possibly called with actual arguments chosen from the monitored expressions M_ℓ of suitable type. Given an expression e, its *top-level subexpressions* $\mathcal{S}_e$ are the expressions corresponding to the nodes at depth 1 in e's abstract syntax tree—namely, the root's immediate children. For example, the top-level subexpressions of `(a + b) < c.d()` are `a + b` and `c.d()`. Then, $\Delta'_{\ell,e} = \bigcup_{s \in \mathcal{S}_e} \Delta_{\ell,e}$ denotes all expressions derived from e's top-level subexpressions.

Modifying the state. For every top-level subexpression e of b, if e is assignable to, JAID generates the fix action e `=` δ for each $\delta \in \Delta'_{\ell,b}$ whose type is compatible with e's.

Modifying an expression. For every top-level subexpression e of b that is not assignable to, but appears in the statement S at ℓ, JAID generates the fix action `tmp_e =` δ`;` $S[e \mapsto$ `tmp_e`$]$ for each $\delta \in \Delta'_{\ell,b}$ whose type is compatible with e's; `tpm_e` is a fresh variable with the same type as e, and $S[e \mapsto$ `tmp_e`$]$ is the statement at ℓ with every occurrence of e replaced by `tmp_e`—which has just been assigned a modified value.

Mutating a statement. "Semantic" fix actions—based on the information captured by the state in suspicious snapshots—are usefully complemented by a few "syntactic" fix actions—based on simple mutation operators that capture common sources of programming mistakes such as off-by-one errors. Following an approach adopted by other APR techniques [6], [9], JAID generates mutations mainly targeting *conditional expressions*. Precisely, if the statement S at ℓ is a conditional or a loop, JAID generates fix actions for every *Boolean* subexpression e of b that appears in the conditional's condition or in the loop's exit condition: 1) if e is a comparison $x_1 \bowtie x_2$, for $\bowtie \in \{$`<`, `<=`, `>=`, `>`$\}$, JAID generates the fix action $S[e \mapsto (x_1 \bowtie' x_2)]$, for every comparison operator $\bowtie' \neq \bowtie$; 2) JAID also generates the fix actions $S[e \mapsto$ **`true`**$]$ and $S[e \mapsto$ **`false`**$]$, where e is replaced by a Boolean constant. In addition to targeting Boolean expressions, if the statement S at ℓ includes a *method call* `t.m(`$\texttt{a}_1, \ldots, \texttt{a}_n$`)`, JAID generates the fix action $S[$`m` $\mapsto$ `x`$]$, which calls any applicable method `x` on the same target and with the same actual arguments as `m` in s.

Modifying the control flow. Even though fix actions may indirectly change the control flow by modifying the state or a branching condition, a number of bugs require abruptly redirecting the control flow. To achieve this, JAID also generates the following fix actions independent of the snapshot information: 1) if method `fixMe` is a procedure (its return type is **`void`**), JAID generates the fix action **`return`**; 2) if method `fixMe` is a function, JAID generates the fix action **`return`** e, for every basic expression of suitable type available at ℓ; 3) if ℓ is a location inside a loop's body, JAID generates the fix action **`break`**.

2.4 Fix Generation: Candidate Fixes

Each fix action—built by JAID as described in Sec. 2.3—is a statement that modifies the program behavior at location ℓ in a way that avoids the state indicated by some suspicious snapshot $s = \langle \ell, b, \texttt{?} \rangle$. In several cases, a fix action should not be injected into the program under repair *unconditionally*, but only when state $b == \texttt{?}$ is actually reached during a computation.

To implement such conditional changes of behavior, JAID uses the *schemas* in Fig. 2 to insert fix actions into the method `fixMe` under repair at location ℓ. In addition to a fix action, some schemas also include the `oldStatement` at location ℓ in the faulty method `fixMe` and the condition b `==` `?` that comes from a snapshot's program state abstraction. Together, the five schemas cover all possible ways of conditionally executing an action, conditionally executing an existing statement, and conditionally executing an action instead of an existing statement. Sec. 4.4 discusses which schemas turned out to be more useful to build correct fixes in our experiments.

During fix generation, JAID first instantiates every applicable schema with a fix action, the `oldStatement`, and the condition determined by the corresponding snapshot; then, it builds *fix candidates* by *replacing* the statement at ℓ in `fixMe` with each instantiated schema.[1]

```
Schema A: action; oldStatement;
Schema B: if (suspicious) { action; } oldStatement;
Schema C: if (suspicious) { oldStatement; }
Schema D: if (suspicious) { action; }
          else { oldStatement; }
Schema E: /* oldStatement; */ action;
```

Fig. 2: Schemas used by JAID to build candidate fixes

2.5 Fix Validation

Even if JAID builds candidate fixes based on a semantic analysis of the program state during passing vs. failing tests, the candidate fixes come with no guarantee of satisfying the tests. To ascertain which candidates are suitable, a fix validation process, which follows fix generation, runs all tests from T that exercise the faulty method `fixMe` against each generated candidate fix. Candidate fixes that pass all the tests are classified as *valid* (also "test-suite adequate" [2] or "plausible" [1]) and retained; other candidates, which fail some tests, are discarded—as they do not fix the fault, they introduce a regression, or both.

2.6 Fix Ranking

Like most generate-then-validate APR techniques, JAID's process is based on heuristics and driven by a finite collection of tests, and thus is ultimately *best effort*: a valid fix may still be incorrect, passing all available tests only because the tests are incomplete pieces of specification. JAID mitigates this problem by *ranking* valid fixes using the same heuristics that underlies fault localization. Every fix includes one fix action, which was derived from a snapshot s; the higher the suspiciousness of s, the higher the fix is ranked; fixes derived from the same snapshot are ranked in order of generation, which means that "semantic" fixes (modifying state or expressions) appear before "syntactic" fixes (mutating statements or modifying the control flow), and fixes of the same kind are enumerated starting from the syntactically simpler ones.

2.7 Revised JAID Implementation

This paper's release of JAID—called "new JAID" in this section—uses the same fundamental techniques and overall design as the previous release [5]—called "old JAID" in this section—but improves several implementation details that help make the tool more widely applicable.

JAID instruments the source code of programs under analysis both to monitor program states and to validate candidate fixes. Old JAID's instrumentation did not work correctly on methods with a **`throws`** clause, which prevented programs using such methods from being analyzed. New JAID extends the instrumentation generation process so that it handles correctly all methods—including those with a **`throws`** clause.

JAID extensively relies on the Java Debug Interface (JDI) to retrieve the value of snapshot expressions (Sec. 2.1) at different program states. To this end, old JAID entirely relied on JDI's *string-based* expression evaluation methods. The advantage of the string-based interface is that it is easier (more uniform) to access since it directly inputs source-code expressions in string form; the disadvantage is that it has some glitches that make expression evaluation crash occasionally—leading old JAID to fail to generate some fixes. To avoid these problems, new JAID primarily uses JDI's *programmatic* API, which supports retrieving object states and evaluating expressions on them directly without going through a string representation. The string-based approach is still available to new JAID, but it is only used as a fallback mechanism for the few cases that cannot be easily handled through the programmatic interface. This makes new JAID's snapshot expression evaluation more robust overall, and hence applicable to more examples.

Finally, new JAID identifies, using hashing, (syntactic) duplicate candidate fixes and immediately discards them, so that the same candidate is not validated twice. This helps trim down validation time by avoiding wasteful effort.

3 EXPERIMENTAL EVALUATION: DESIGN

We experimentally evaluated JAID on 693 faults from three curated collections often used in automated program repair research. These faults come from Java applications and libraries of different size and complexity in disparate domains; this makes the results of JAID's evaluation more likely to be representative of its general behavior. At the same time, using faults from standard benchmarks makes it possible to meaningfully compare JAID with the other state-of-the-art APR tools for Java.

3.1 Research Questions

The experiments address the following research questions.

RQ1: How *effective* is JAID? In RQ1, we evaluate the effectiveness of JAID from a user's perspective—in terms of the number of faults that JAID can fix.

RQ2: How *efficient* is JAID? In RQ2, we examine the running time of JAID on different bugs.

RQ3: How effective is JAID in comparison to *other APR techniques* for Java? In RQ3, we directly compare JAID to several other state-of-the-art APR tools for Java.

RQ4: Do all *templates* and *heuristics* have an *impact* on JAID's effectiveness? In RQ4, we zoom in on the relative usefulness of various kinds of templates and heuristics used by JAID's APR algorithm.

RQ5: How sensitive is JAID's behavior to the choice of *fault localization* algorithm? In RQ5, we assess whether replacing JAID's custom fault localization algorithm with a different one affects its overall effectiveness.

3.2 Subjects

Our evaluation uses 693 faults from three widely used benchmarks of Java bugs: DEFECTS4J [2], INTROCLASSJAVA [10], and QUIXBUGS [11].

1. Since each fix generated by JAID combines one fix action and one schema, it adds at most 4 new lines of codes to a patched method.

TABLE 1: DEFECTS4J benchmark: how many BUGS, TESTS, and thousands of lines of code (KLOC) DEFECTS4J includes from each PROJECT.

PROJECT	DESCRIPTION	KLOC	TESTS	BUGS
Chart	JFreechart	96	2205	26
Closure	Closure Compiler	90	7927	133
Lang	Apache Commons-Lang	22	2245	65
Math	Apache Commons-Math	85	3602	106
Time	Joda-Time	27	4130	27
Mockito	Mockito	43	1161	38
	TOTAL	320	20109	395

TABLE 2: INTROCLASSJAVA benchmark: for each PROGRAM, how many **b**lack-box and **w**hite-box TESTS exercise it, and how many of its BUGS are triggered by the **b**lack-box and the **w**hite-box tests.

			TESTS		BUGS	
PROGRAM	DESCRIPTION	LOC	**b**	**w**	**b**	**w**
checksum	checksum of a string	18	6	10	7	11
digits	digits of a number	13	6	10	51	60
grade	grade from score	22	9	9	89	88
median	median of 3 numbers	24	7	6	51	48
smallest	min of 4 numbers	24	8	8	47	45
syllables	count vowels	17	6	10	13	12
	TOTAL	118	42	53	258	264
	detectable unique bugs in $\mathbf{b} \cup \mathbf{w}$:				297	

TABLE 3: QUIXBUGS benchmark: the MIN, MEDIAN, MAX, and TOTAL *size* of the faulty methods (in lines of code), and number of *tests* targeting each method.

	MIN	MEDIAN	MAX	TOTAL
Size (LOC)	2	17	45	717
Tests	3	6	13	259

DEFECTS4J (revision #895c4e6) includes 395 bugs from 6 projects: Chart, Closure, Lang, Math, Time, and Mockito; Tab. 1 displays basic statistics about these projects in DEFECTS4J. The bugs included in DEFECTS4J are a diverse sample of real-world bugs, and as such they include both several that admit simple fixes and others that require changes that span multiple methods or even multiple files.

Each bug in DEFECTS4J has a unique identifier, corresponds to a buggy version and a programmer-fixed version of the code, and is accompanied by some programmer-written unit tests that exercise the code—in particular, at least one test triggers a failure on the buggy version. Our experiments used all the 395 bugs in DEFECTS4J.

Our previous work [5] ran JAID on a pre-selected subset of DEFECTS4J bugs, excluding for simplicity those that were likely outside JAID's capabilities. In the present paper, we use instead *all* the bugs in DEFECTS4J as our subjects. The main reason for doing so is that using all available bugs in a benchmark provides the most comprehensive data about JAID's performance, and hence makes for a fairer comparison with other tools evaluated on the same benchmark. For example, measures such as average running time should uniformly include the time to run a tool on all bugs, including those where the tool will fail to produce any fixes. Including all bugs is also the recommendation of all large-scale replication studies [2], [12], [13]. A more specific reason for not pre-selecting bugs is that sometimes there is no sure way to know in advance on which bugs JAID will be successful. For example, Sec. 4.1 examines five "hard" bugs that JAID unexpectedly managed to fix with fixes that were semantically equivalent to syntactically much more complex fixes written by developers.

INTROCLASSJAVA is a direct Java translation [10] of the IntroClass benchmark [14], which collects several buggy student-written solutions to six small assignments given in an introductory, undergraduate programming course. As such, programs in INTROCLASSJAVA are a meaningful sample of the kinds of mistakes commonly made by novice programmers writing small programs. Tab. 2 displays basic statistics about the 297 buggy programs translated to Java in INTROCLASSJAVA.

Each program in INTROCLASSJAVA comes with two JUnit test suites (also translated from C): a black-box one written by the instructor based on the program's specification, and a white-box one generated using a symbolic execution tool. Following the practice of other studies [14]–[16], JAID had access only to the black-box tests; the white-box tests feature only in the experimental evaluation, where we used them to determine which of the fixes outputted by JAID were correct. Since 39 buggy programs in INTROCLASSJAVA do not cause any black-box tests to fail (that is, only white-box tests trigger failures), we excluded them from the experiments and used the remaining 258 program variants as subjects in the experiments.

QUIXBUGS contains 40 faulty programs taken from the Quixey Challenge [11], where programmers had one minute to produce a fix given an implementation of a classic algorithm with a bug on a single line. The algorithms include classics such as Dijkstra's shortest path on a graph, building the minimum spanning tree, and sorting algorithms; therefore, the buggy programs in QUIXBUGS are representative of programming mistakes that are commonly made when implementing algorithms that are challenging to get right. Tab. 3 displays basic statistics about the programs in QUIXBUGS.

Each faulty program in QUIXBUGS comes with a correct reference implementation, as well as passing and failing tests; thus, the textual diff between a buggy program and the corresponding reference implementation plays the role of the programmer-written fix for the bug. Our experiments used all the 40 bugs in QUIXBUGS, together with all available tests.

3.3 Other Automated Program Repair Tools for Java

We quantitatively compared JAID to 16 tools for APR of Java programs; we considered all tools working on Java that used at least one of the three benchmarks (DEFECTS4J, INTROCLASSJAVA, and QUIXBUGS) in their published experimental evaluation. Tab. 4 lists the 16 tools;[2] if an experimental evaluation of tool T on benchmark B is publicly available, the table indicates, at row T and column B, the source of

2. SketchFixPP is a variant of the SketchFix technique discussed in the same paper [17]

the experimental results that we used in the comparison with JAID.

While most tools use DEFECTS4J as benchmark in their experiments, it is possible that the experimental evaluations of different tools select different *subsets* of bugs in DEFECTS4J. This may happen for two reasons: first, since the DEFECTS4J benchmark is occasionally extended with new projects, it may happen that an experimental evaluation on an older version of DEFECTS4J simply did not have access to some bugs that are available in DEFECTS4J at the time of writing. Second, an experimental evaluation may deliberately exclude some bugs from the experiments if the technique being evaluated or its implementation are not easily applicable to those bugs. For example, [2] excludes project Closure because of difficulties in executing its tests; and the experiments in the original paper on JAID [5] included only bugs whose programmer-written fix modifies at most 5 consecutive lines of code—since JAID was unlikely to succeed on bugs requiring more complex fixes.

Despite these discrepancies, our quantitative comparison with other tools is sound:

- Since bugs that are excluded *a priori* from an evaluation can normally be considered beyond a tool's current capabilities, we compute *recall* relative to the same largest set of DEFECTS4J bugs, which is a superset of all those used in any evaluations. In contrast, excluding bugs from an evaluation does not negatively affect a tool's *precision*—in fact, it puts JAID at a disadvantage because we insisted on running it on every available bug in DEFECTS4J.
- We report absolute numbers of valid and correct fixes, as well as the bugs that each tool *uniquely* fixes.
- We ascertain that the version of DEFECTS4J used in our experiments does not differ substantially, on the same bugs, from those used in the other tools' experiments.

We did not perform any quantitative comparison of other measures, running time in particular, as these would require to replicate experiments with other tools with common settings—which is not always possible or easy, since not all publications come with a complete replication package. Since improving performance is not a primary concern in automated program repair research at the moment, and publications rarely emphasize performance data, the comparison of JAID with other tools focuses on metrics that are generally accepted as practically interesting.

3.4 Fault-localization techniques

JAID employs a custom spectrum-based [24] fault localization technique, while most other APR tools directly reuse standard fault localization approaches. To find out whether JAID's behavior depends significantly on how fault localization works (and hence to answer RQ5), we ran experiments where JAID uses other well-known spectrum-based fault localization techniques instead of its custom algorithm.

Tab. 5 lists the five spectrum-based techniques [25] used to replace JAID's. All of them work by computing a *suspiciousness* score of each program entity (in the case of JAID, *snapshots* which include a location and part of the state—as described in Sec. 2.2) based on its coverage by passing and failing tests.

TABLE 4: Automated program repair tools for Java used in the comparison with JAID. For each benchmark, the table reports the source of a tool's evaluation results used for a quantitative comparison with JAID; no reference means the tool has not been evaluated on the benchmark.

TOOL	DEFECTS4J	INTROCLASSJAVA	QUIXBUGS
ACS	[7]		
Astor			[18]
CapGen	[16]	[16]	
Elixir	[19]		
HAD	[6]		
JFix		[15]	
jGenProg	[2]		
jKali	[2]		
Nopol	[2]		[20]
S3		[21]	
SimFix	[22]		
SketchFix	[17]		
SketchFixPP	[17]		
ssFix	[23]		
xPar	[6], [7]		

TABLE 5: Spectrum-based fault localization ALGORITHMs used to replace JAID's. Each algorithm defines a formula to compute the SUSPICIOUSNESS of any state snapshot s as a function of $\mathsf{P}(s)$ (the number of passing tests where s is observed) and $\mathsf{F}(s)$ (the number of failing tests where s is observed), and relative to the total number of passing (P) and failing (F) tests.

ALGORITHM	SUSPICIOUSNESS OF SNAPSHOT s
Tarantula [26]	$\frac{\mathsf{F}(s)/\mathsf{F}}{\mathsf{F}(s)/\mathsf{F} + \mathsf{P}(s)/\mathsf{P}}$
Ochiai [27]	$\frac{\mathsf{F}(s)}{\sqrt{\mathsf{F}\cdot(\mathsf{F}(s) + \mathsf{P}(s))}}$
Op2 [28]	$\mathsf{F}(s) - \frac{\mathsf{P}(s)}{\mathsf{P}+1}$
Barinel [29]	$1 - \frac{\mathsf{P}(s)}{\mathsf{P}(s) + \mathsf{F}(s)}$
DStar [30] (with $* = 2$)	$\frac{\mathsf{F}(s)^*}{\mathsf{P}(s) + \mathsf{F} - \mathsf{F}(s)}$

3.5 Experimental Setup

All the experiments ran on a cloud infrastructure, with each run of JAID using exclusively one virtual machine instance, configured to use one core of an Intel Xeon Processor E5-2630 v2, 8 GB of RAM, Ubuntu 14.04, and Oracle's Java JDK 1.8.

Each experiment targets one subject bug k, and runs JAID with buggy code bug_k and tests T_k as input; the output is a ranked list of valid fixes for the bug. The process to determine which of JAID's fixes are *correct* follows standard research practices:

- The fix of a bug in DEFECTS4J or QUIXBUGS is correct if manual inspection convincingly indicates that it is *semantically equivalent* to the programmer-written fix correct_k. We allot around 5 minutes per fix to determine semantic equivalence; if we cannot conclude that the fix is equivalent after 5 minutes, we classify it as incorrect. This conservative assessment implies that a fix classified as correct is a fix suggestion that could have been deployed (or is very close to one that could have been deployed).
- The fix of a bug in INTROCLASSJAVA is correct if it passes all available white-box tests (which JAID had

not access to), in addition to the black-box tests that JAID uses directly for validation. Programs and bugs in INTROCLASSJAVA are sufficiently simple that passing all white-box tests provides high confidence in their correctness.

We did not set a *time limit* in our experiments: since JAID ranks all generated snapshots according to their suspiciousness (see Sec. 2.2), and depends on the ranking to guide the following stages, setting an arbitrary cutoff time may prevent JAID from generating a complete ranking. Instead, we limited the search space in our experiments by configuring JAID so that it uses at most 1500 snapshots in order of suspiciousness; then, the following stages (Fig. 1) all run to completion. The number 1500 was chosen heuristically; some of the experiments reported in Sec. 4.4 indicate that this choice achieves a reasonable trade-off, but also that JAID's overall performance is fairly robust with respect to the choice of how many snapshot to process.

4 EXPERIMENTAL EVALUATION: RESULTS

This section presents the results of the experimental evaluation of JAID carried out according to the design of Sec. 3. Averages are measured using the *median* by default, with exceptions explicitly pointed out.

4.1 RQ1: Effectiveness

Tab. 6 displays the key results of the experimental evaluation of JAID's effectiveness. As shown there, JAID produced *valid* fixes for 189 bugs in total: 94 bugs in DEFECTS4J, 84 bugs in INTROCLASSJAVA, and 11 bugs in QUIXBUGS. More significantly, it produced *correct* fixes for 113 bugs: 35 in DEFECTS4J, 69 in INTROCLASSJAVA, and 9 in QUIXBUGS. These numbers correspond to an overall:

precision: 59.5% the percentage of bugs with a valid fix for which JAID outputs at least a correct fix

recall: 15.4% the percentage of all bugs for which JAID outputs at least a correct fix

As we discuss in Sec. 4.3, JAID's effectiveness is competitive with the state of the art; JAID is capable of producing fixes of high quality in a significant number of cases on code of various complexity and maturity level.

Precision and recall are much higher with benchmarks INTROCLASSJAVA and QUIXBUGS than with DEFECTS4J. This difference is likely the result of two aspects. First, DEFECTS4J programs are often larger and require more complex fixes than those in the other two benchmarks. Second, tests in INTROCLASSJAVA and QUIXBUGS are more thorough, and hence provide stronger specifications of the intended program behavior, and support a more effective test-based validation that prunes out valid but incorrect fixes.

Still, JAID can be effective even with weak oracles: it correctly fixed 5 bugs in DEFECTS4J that include only one failing test (and no passing tests), ranking the correct fix first in two cases.

Among the 35 bugs from DEFECTS4J that JAID correctly fixes, 5 are from project Chart, 13 from project Closure, 6 from project Lang, 9 from project Math, 1 from project Time, and 1 from project Mockito. In particular, Mockito bugs were not used to evaluate most other APR tools. A recent study [31] shows that Nopol can correctly fix 1 bug while SimFix and CapGen cannot propose any valid fix to bugs in Mockito even if all the buggy locations are analyzed. Compared with that, JAID generated valid fixes to 4 Mockito bugs (and correct fixes to 1).

When JAID is ineffective. To better understand the limitations of JAID, we manually analyzed the bugs where JAID failed to produce any valid fix. In the analysis, we restrict ourselves to the faults from DEFECTS4J and QUIXBUGS, since only bugs in these two benchmarks are accompanied by programmer-written fixes. We also excluded from this manual analysis faults whose programmer-written fixes:

- modify more than one method (108 bugs excluded);
- modify source code outside the only buggy method (e.g., class fields and/or `import` statements) (13 bugs excluded); or
- modify more than four lines of code, according to the dissection data of DEFECTS4J [32] (99 bugs excluded).

Faults whose fixes have these characteristics are clearly outside the current capabilities of JAID's algorithm,[3] and hence it is unsurprising that JAID failed to produce any fix for them.

Examining the remaining 104 bugs that JAID could not fix, we identified three phases from which JAID's ineffectiveness commonly originates (see Tab. 7 for a summary of the number of bugs in each category):

Setup: JAID uses a simple Python script to automatically extract project configuration information from the DEFECTS4J framework. The script works well on most DEFECTS4J bugs, but when it fails to properly setup the environment it prevents JAID from running at all.
An unsuccessful setup prevented 36 bugs from being fixed; these failures are the result of practical limitations of the current implementation of JAID, but they do not reflect any fundamental limitation of JAID's approach.

Fault localization: JAID's repair process uses fault localization as a crucial starting point: when the fault localization algorithm fails to track the relevant failing condition in a snapshot expression, or ranks the corresponding snapshot past the cutoff, the following fix generation step cannot be successful because it does not have the necessary ingredients.
Ineffective fault localization prevented 49 bugs from being fixed. Even though JAID uses only the 1500 most suspicious snapshots for fix generation, the cutoff is not a significant limitation in practice: if we increase it to allow 3000 snapshots, JAID correctly fixes 1 more bug (`Lang39`); if we remove the cutoff entirely, JAID correctly fixes 2 more bugs (`Closure104` and `Lang24`). Thus, the current cutoff value is a reasonable balance between costs and effectiveness; and improving the effectiveness of fault localization in JAID is likely to require more radical changes to the kind of information it uses. We discuss further details about the effectiveness of fault localization in JAID in Sec. 4.5.

3. We could trivially lift some of these limitations, for example by supporting the generation of fixes that combine multiple actions. However, such changes would make the fix space much larger, without also providing a scalable mechanism to explore it efficiently. Thus, effectively lifting these limitations would require modifications of the approach that go beyond simple extensions of the search space.

TABLE 6: Effectiveness of JAID: main experimental results for each benchmark DEFECTS4J, INTROCLASSJAVA, and QUIXBUGS, as well as OVERALL (bottom rown in each table).

(a) Number of BUGS for which JAID produced VALID FIXES; number of ALL valid fixes produced for all bugs; and MEDIAN number of valid fixes produced per bug for which at least a valid fix was produced.

	VALID FIXES		
BENCHMARK	BUGS	ALL	MEDIAN
DEFECTS4J	94	16762	23.5
INTROCLASSJAVA	84	4243	28.5
QUIXBUGS	11	2187	10.0
OVERALL	189	23192	22.0

(b) Number of bugs for which JAID produced CORRECT FIXES in ANY position, FIRST position, a TOP-10 position; median POSITION of the first correct fix in JAID's output; and number of ALL correct fixes produced for all bugs; and the MEDIAN number of correct fixes produced per bug for which at least a correct fix was produced.

	CORRECT FIXES					
BENCHMARK	ANY	FIRST	TOP-10	POSITION	ALL	MEDIAN
DEFECTS4J	35	14	25	3	139	2
INTROCLASSJAVA	69	30	43	2	829	5
QUIXBUGS	9	4	9	2	22	2
OVERALL	113	48	77	2	990	3

(c) Precision and recall obtained by JAID. PRECISION is the ratio $100 \cdot \text{ANY}/\text{BUGS}$ denoting the percentage of bugs with a valid fix that is also correct; and RECALL is the ratio $100 \cdot \text{ANY}/\text{TOTAL}$ denoting the percentage of TOTAL bugs with a correct fix.

BENCHMARK	PRECISION	RECALL
DEFECTS4J	37.2	8.9
INTROCLASSJAVA	82.1	26.7
QUIXBUGS	81.8	22.5
OVERALL	59.5	15.4

TABLE 7: Bugs in DEFECTS4J and QUIXBUGS for which JAID failed to produce any valid fix. The total number of such bugs (ALL) is split according to where JAID's ineffectiveness originates: in an unsuccessful SETUP, in an ineffective FAULT LOCALIZATION, or in a limited FIX SPACE. Some bugs are the combined result of ineffective fault localization and limited fix space, and thus are counted in both columns.

	FAULTS NOT FIXED			
	ALL	SETUP	FAULT LOCALIZATION	FIX SPACE
DEFECTS4J	104	33	41	63
QUIXBUGS	29	3	8	24
TOTAL	133	36	49	87

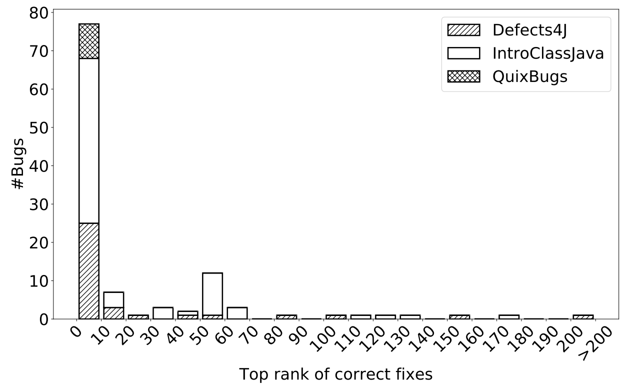


Fig. 3: Distribution of the top rank of correct fixes of bugs.

Fix space: even with perfect fault localization, some fixes may require program modifications that are inexpressible using JAID's fix actions and schemas, and hence fall outside its fix space.

A limited fix space prevented 87 bugs from being fixed. Examples of features outside JAID's fix space that were used in some human-written fixes include: **new** expressions with type casting; calls to methods with arguments; constants declared in specific classes; adding a **case** to an existing **switch** statement; adding an **if** with a compound then or else block; adding a compound statement (such as a **try/catch** block, or a loop).

While extending JAID so that it generates such complex statements as fix actions is not technically complex, the expanded fix space would become too large to effectively search in a reasonable time using JAID's heuristics. Extending JAID's search space selectively—focusing on fixing only certain categories of bugs—is an interesting direction for future work; but JAID's current focus is on being "general-purpose".

Ranking of fixes. When it is successful, JAID often produces several valid fixes—122.7 (= 23192/189) per fixed bug on average in our experiments—and a much smaller number of correct fixes—8.8 (= 990/113) per fixed bug on average in our experiments. When JAID outputs several valid fixes for the same bug, it ranks them according to a simple heuristics that tries to put on top those that are more likely to be correct.

The stacked histogram in Fig. 3 depicts the distribution of the top ranks of correct fixes. Among the 113 bugs that JAID correctly fixed, the median position of the first correct fix was 2 (from the top); the correct fix appeared at the top position in 48 bugs; and among the top-10 valid fixes for 77 bugs. For the remaining 36 bugs that were correctly fixed, the correct fix appears further down in the output list; in 26 of these cases, the correct fix turns out to be a "syntactic" one, but several valid, incorrect "semantic" fixes are generated and ranked higher.[4]

These results indicate the importance of ranking to ensure that the correct fixes are easier to spot among several valid but incorrect ones. JAID's ranking heuristics often do a good job, but there is room for improvement. In particular, ranking may benefit from mining additional information about common features of programmer-written fixes, as was done in the other repair tools like ACS, HDA, SimFix, and CapGen.

Syntactic complexity of fixes. In DEFECTS4J and QUIXBUGS, we classify a fix as correct if it is *semantically* equivalent to the fix written by programmers for the same

4. Sec. 2.3 explains what syntactic and semantics fix actions are. Sec. 4.4 gives detailed statistics on the usage of different types of fix actions in generated correct fixes.

bug. A potential risk is that JAID generates fixes that are semantically equivalent but *syntactically* very different—in particular, needlessly complex and less readable. In practice, this is unlikely to happen because JAID cannot generate fixes that are syntactically very complex; for the simpler fixes it can generate, semantic and syntactic equivalence tend to coincide.

There are a few exceptions to this general observation: the programmer-written fixes for 5 of the 35 DEFECTS4J bugs that JAID can correctly fix involve modifications of multiple lines at one or several locations, or even deletion of a whole method. Thus, JAID's correct fixes for these bugs were syntactically much simpler than, but still semantically equivalent to, the programmer-written fixes. On the one hand, this is encouraging because it proves the flexibility of JAID's repair algorithm. On the other hand, one could argue that the more complex programmer-written fixes are in some cases preferable because they are refactoring aspects of the program's design in order to improve the code beyond the single functionality that is being repaired.

For example, the programmer-written fix of bug Closure 46 involves deleting a 17-line long method `getLeastSupertype` from class `RecordType`. Method `getLeastSupertype` overrides the method with the same name in superclass `PrototypeObjectType`, so the fix replaces, implicitly by inheritance, all invocations to the overriding method `RecordType.getLeastSupertype` with calls to the overridden one `PrototypeObjectType.getLeastSupertype`. Instead of removing method `RecordType.getLeastSupertype`, JAID fixes the same bug by changing `RecordType.getLeastSupertype`'s body so that it explicitly calls the overridden method `PrototypeObjectType.getLeastSupertype` as **`super`**`.getLeastSupertype` and just returns the call's result. In this case, the programmer-written fix has a better design since it does not leave dead code in the repaired program, but the fix produced by JAID is still completely behaviorally correct and hence practically useful.

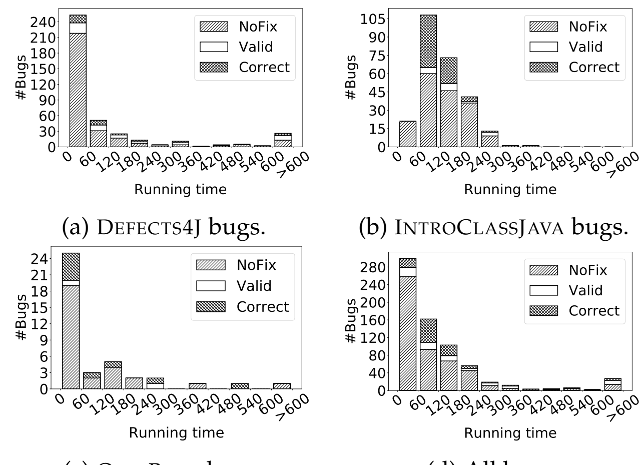
JAID *produced a correct fix for over 15% of all bugs it analyzed, achieving a precision of nearly 60%. It ranked most correct fixes high in the output list (in position 2 on average).*

4.2 RQ2: Efficiency

We measure the efficiency of JAID in terms of *running time*: overall running time, and running time until the first fix is generated.

Overall running time. Fig. 4 shows the distribution of JAID's overall running time in every experiment. The distribution is clearly skewed to the left, indicating that the running time of JAID on most bugs is limited. For example, JAID ran for up to 60 minutes on about 43% of all bugs, and for up to 240 minutes on about 89% of all bugs.

JAID's running time distribution for the INTROCLASSJAVA benchmark looks different from the other benchmarks: in the latter, JAID ran for less than 60 minutes on the majority of bugs; but in INTROCLASSJAVA, the most frequent running times are between 60 and 120 minutes. This is counterintuitive given that programs in INTROCLASSJAVA are generally simpler and smaller than those in the other benchmarks. We found out this behavior is due to a feature of programs in



(a) DEFECTS4J bugs. (b) INTROCLASSJAVA bugs.

(c) QUIXBUGS bugs. (d) All bugs.

Fig. 4: Distribution of JAID's running time per bug. The height of each bar spanning x coordinates a and b marks the total number of bugs whose overall fixing time was between a and b minutes. Segments of different hatch partition the number of bugs into those for which JAID produced no fixes, only valid fixes, or correct fixes.

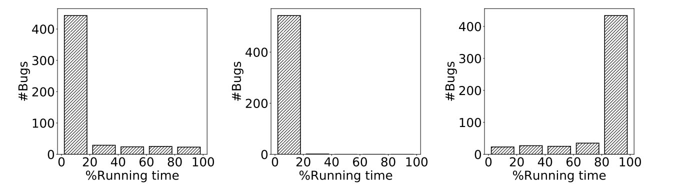
TABLE 8: Statistics on JAID's overall running time per bug (in minutes): MINimum, MAXimum, MEAN, MEDIAN, STandard DEViation, and SKEWness. Each statistics is computed over ALL bugs, bugs with a VALID fix, and bugs with a CORRECT fix.

	MIN	MAX	MEAN	MEDIAN	STDEV	SKEW
ALL	0.0	3403.6	136.3	76.3	283.8	6.3
VALID	0.6	3403.6	194.0	95.3	356.4	5.5
CORRECT	2.1	1777.7	154.5	92.8	252.0	4.7

INTROCLASSJAVA, which all use a `Scanner` object to read user input from the console. Since the `Scanner` class defines many observer methods of the kinds used by JAID's heuristics to construct snapshot expressions and to generate fix actions (see Sec. 2.1 and Sec. 2.3), JAID generated more than 10,000 candidate fixes for several bugs in INTROCLASSJAVA, thus leading to a longer cumulative running time.

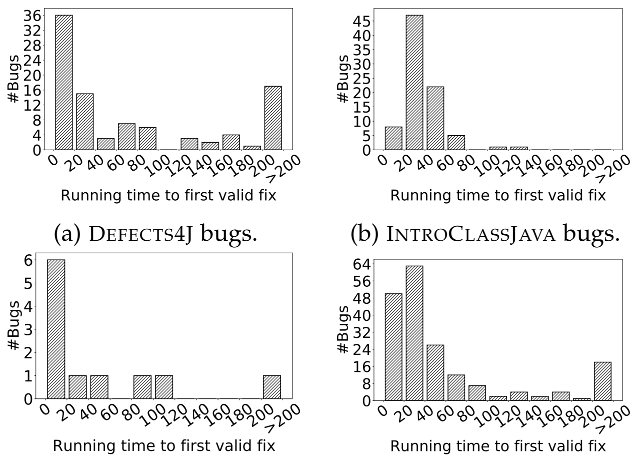
The overall running times of JAID are not short in absolute terms, but they are to be expected in a tool that relies extensively on *dynamic analysis*, which requires to repeatedly execute several variants of heavily instrumented programs. JAID's performance is consistent regardless of whether the bugs will eventually be fixed: if we look at only bugs that JAID could correctly fix, it ran for up to 100 minutes on about 55% of them, and for up to 200 minutes on about 88% of them. In all, these running times are suitable for JAID's intended mode of usage as a batch (not interactive) analysis tool.

Tab. 8 breaks down statistics on the overall running time per bug into all sessions, those that produced a valid fix, and those that produced a correct fix. JAID ran in around 76 minutes per any bug on average; and in around 95 minutes per bugs on which it is successful (producing a valid or correct fix). JAID is unsurprisingly significantly slower than tools based on constraint solving and other static techniques (see Sec. 6); for example, Nopol [9] takes around 22 minutes



(a) Fault localization. (b) Fix generation. (c) Fix validation.

Fig. 5: Distributions of JAID's running time per bug in each main phase of the fixing process (fault localization, fix generation, and fix validation). The height of a bar spanning x coordinates a and b denotes the number of bugs whose running time in that phase (fault localization, fix generation, and fix validation) was between a% and b% of the overall running time. The 36 faults whose fixing failed during setup phase (Sec. 4.1) are excluded from the figure.



(a) DEFECTS4J bugs. (b) INTROCLASSJAVA bugs. (c) QUIXBUGS bugs. (d) All bugs.

Fig. 6: Distribution of JAID's running time until it finds a valid fix over all bugs for which JAID finds at least a valid fix. The height of a bar spanning x coordinates a and b marks the number of bugs whose running time until JAID output a valid fix was between a and b minutes.

TABLE 9: Statistics on JAID's running time (in minutes) until a VALID or a CORRECT fix is found: MINimum, MAXimum, MEAN, MEDIAN, STandard DEViation, and SKEWness. The statistics are computed over bugs with a VALID fix (row VALID), and bugs with a CORRECT fix (row CORRECT).

	MIN	MAX	MEAN	MEDIAN	STDEV	SKEW
VALID	0	1777	45.4	2	180.2	7.6
CORRECT	0	1777	78.7	24	221.0	6.4

per bug on average—on what, we assume, is comparable hardware. JAID's running time are, however, in line with other techniques mainly based on dynamic analysis—such as jGenProg [2] which takes about one hour per bug.

There is room for improving JAID's performance by fine-tuning its implementation. To better understand the main performance bottlenecks, Fig. 5 breaks down the running time into the different phases of JAID's overall process: fault localization, fix generation, and fix validation. *Fix validation* remains, by far, the most time consuming phase, taking about 76% of the overall running time on all the bugs. A straightforward way to practically reduce this is to run validation of candidates in parallel, which we plan to implement in future versions of JAID.

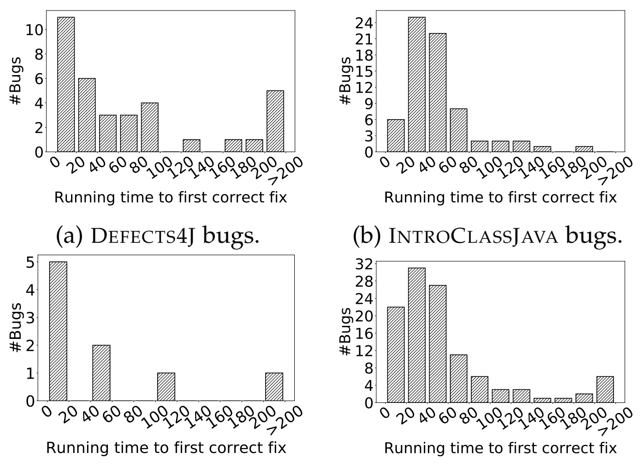
The other phases of JAID take proportionally much less time: fault localization accounts for no more than 20% of the running time for 443 faults, which indicates JAID's fault localization is reasonably efficient on most faults. In a minority of cases, however, fault localization may become a bottleneck: JAID's fault localization took over 60% of overall running time for 48 faults that required many snapshot expressions, had many tests, or both. One way of improving the efficiency of fault localization in these cases is therefore to *select* snapshot expressions and tests (for example, based on information gathered during fixing). As for the time spent on fix generation, it never takes more than 15% of the overall running time.

Time to first fix. In its current implementation, JAID always runs to completion, stopping only after it has validated all the generated candidate fixes. From a user's point of view, however, what matters is the time JAID takes to generate the *first valid* or the *first correct* fix: as soon as a valid fix is available, the user can start inspecting it to see if it's suitable; and as soon as a correct fix is available, the rest of JAID's output becomes redundant.

To assess JAID's efficiency from this more practical viewpoint, Fig. 6 and Fig. 7 picture the distributions of running time to a valid fix and to a correct fix, and Tab. 9 provides related statistics. The average running time to a valid fix is less than 30 minutes for most faults, and hence the distributions in Fig. 6 are strongly skewed to the left. A similar trend is visible in Fig. 7 for the time to a correct fix, even though the skewedness is less pronounced in this case: while the majority of correct fixes can be produced in less than 90 minutes, there remains a "long tail" of correct fixes that take considerably more time.

JAID validates candidate fixes in the same order of suspiciousness value of their snapshots: the more suspicious a snapshot, the earlier its derived candidate fixes are validated. In contrast, other automated program repair techniques [6], [22] adjust the validation order based on information obtained by mining other fixes and the relations between different fixes. JAID could add a similar prioritization approach to its validation phase. Specifically, it could associate a priority order based on the relation between, on the one hand, fix actions (Sec. 2.3) and fix schemas (Sec. 2.4), and, on the other hand, the characteristics of the bug being fixed in comparison with the features of other known bugs. For example, if fixing bugs involving null-pointer dereferencing usually requires to execute a call only conditionally on its target being non-null, Jaid's schema B in Fig. 2 should be given higher priority. We may experiment with such prioritization approaches in future extensions of JAID.

On an average bug, JAID *ran for around 76 minutes in total, but took only 2 minutes to produce the first valid fix, and 24 minutes to produce the first correct fix (when it could find one).*



(a) DEFECTS4J bugs. (b) INTROCLASSJAVA bugs.

(c) QUIXBUGS bugs. (d) All bugs.

Fig. 7: Distribution of JAID's running time until it finds a correct fix over all bugs for which JAID finds at least a correct fix. The height of a bar spanning x coordinates a and b marks the number of bugs whose running time until JAID output a correct fix was between a and b minutes.

4.3 RQ3: Tool comparison

Tab. 10 quantitatively compares JAID to 16 other APR tools for Java on bugs from the three benchmarks.

JAID can produce *valid* fixes for several more bugs than any other tools; in DEFECTS4J, for example, JAID produced valid fixes for 56% ($= (94 - 60)/60$) more bugs than ssFix, which does so for 60 bugs. This indicates that JAID explores a large space of possible fixes—arguably larger than other tools. JAID outperforms any other tools also in terms of *correct* fixes, even though JAID's advantage is more limited here; in DEFECTS4J, for example, JAID's recall is 0.3% higher (1 more bug correctly fixed) than the runners up SimFix and SketchFixPP, and it is 2.3% higher (9 more bugs correctly fixed) than the next-best tool Elixir.

When it comes to *precision*, several other tools perform better than JAID on DEFECTS4J bugs, especially if we only consider correct fixes that are ranked high (in the top-10 or even in first position). JAID's precision is better than ssFix's, Nopol's, jKali's, and jGenProg's, but it is worse than the other tools for which this measure is available. All tools that outperform JAID in terms of precision are specifically designed to achieve a high precision, and have access to information mined from existing human-written patches to help identify candidate fixes that are more likely to be correct. JAID's precision may similarly improve if its algorithm also had access to the same kind of machine-learned information.

JAID's precision is lagging behind other tools only on DEFECTS4J bugs; in contrast, it is consistently *higher* than other tools on INTROCLASSJAVA[5] and QUIXBUGS bugs. This difference is probably due to the kinds of tests that accompany bugs in INTROCLASSJAVA and QUIXBUGS: thorough tests, which serve as strong oracles. JAID's validation is entirely based on dynamic analysis, and hence using high-quality tests lowers overfitting and increases precision.

JAID's effectiveness on INTROCLASSJAVA and QUIXBUGS also suggests that its repair algorithm may be less prone to *overfitting* than other tools that are more focused on achieving a high precision on DEFECTS4J. A recently published large-scale experiment [12] targeted 11 repair tools for Java (ARJA, two implementations of GenProg for Java, two implementations of Kali for Java, an implementation of RSRepair for Java, Cardumen, jMutRepair, Nopol, DynaMoth, and NPEFix) that were primarily evaluated on DEFECTS4J in their original publications, and ran them on different benchmarks—including INTROCLASSJAVA and QUIXBUGS. While these experiments only considered the number of *valid* fixes (without analyzing correctness), they clearly showed that the number of bugs with valid fixes for "all [11] tools is significantly higher for bugs from DEFECTS-4J compared to the other four benchmarks"—which include INTROCLASSJAVA and QUIXBUGS. This is in contrast to JAID, which built proportionally *more* valid (and correct) fixes for bugs in other benchmarks: in [12]'s experiments, none of the 11 tools could build valid fixes for more than 10% of bugs in INTROCLASSJAVA and QUIXBUGS; in our experiments, JAID built valid fixes for 28% of bugs in both INTROCLASSJAVA and QUIXBUGS.

Unique fixes. Tools that have been evaluated on the DEFECTS4J benchmark often list the identifiers of the bugs they correctly fixed; therefore, we can get a more nuanced idea of which bugs each tool can fix. Fig. 8 displays this information in a readable format; and column UNIQUE in Tab. 10 summarizes the number of bugs that each tool can fix that no other tools can. This data suggests that tools are often *complementary* in the specific bugs they are successful on: JAID fixes 11 bugs that no other tool can fix; SimFix fixes another 12; ACS fixes 11; CapGen, SketchFixPP, and HDA fixes 3 each; Nopol fixes 2; ssFix and jGenProg fixes 1 each. The complementarity between JAID and other tools is not accidental but depends on different tools focusing on different fix spaces and fix ingredients. Consider the 23 bugs that only ACS (11 bugs) or SimFix (12 bugs) can fix: 9 require inserting statements that throw appropriate exceptions, which is something only ACS can do at the moment; 4 require changing the code at two locations simultaneously, which is something very few tools other than SimFix are capable of; 1 requires inserting a snippet with more than four lines of code, which is beyond the capabilities of most tools; and the remaining 9 require various other ingredients currently outside JAID's fix space. The complementarity implies that each technique is successful in its own domain, and suggests that *combining* techniques based on mining (such as SimFix, CapGen, HDA, and ACS) with JAID's techniques is likely to yield further improvements in terms of overall effectiveness.

Other tools. We cannot quantitatively compare APR tools that target other programming languages since they were evaluated on different benchmarks [14]. Nevertheless, just to give an idea, Angelix [33] and Prophet [34] achieve a precision of 35.7% and 42.9%, and a recall of 9.5% and 17%, on 105 bugs in the C GenProg benchmark [35]; AutoFix [36] achieves a precision of 59.3% and a recall of 25% on 204 bugs from various Eiffel projects with contracts. GenProg,

5. While precision measures of other tools on INTROCLASSJAVA bugs are not available, JAID's precision is quite high in absolute numbers (over 82%), and hence it is likely to be overall competitive.

TABLE 10: A quantitative comparison of JAID with 16 other tools for automated program repair, based on their published experimental evaluations (see Tab. 4 for sources of the comparison data), and partitioned into sections for each benchmark DEFECTS4J, INTROCLASSJAVA, and QUIXBUGS. For each APR TOOL, the table reports the number of bugs that the tool could fix with a VALID fix; the number of bugs that the tool could fix with a CORRECT fix; and the resulting PRECISION (CORRECT/VALID) and RECALL (CORRECT/TOT, where TOT is the total number of bugs from the benchmark that are used in the experiments). For tools whose data about the POSITION of fixes in the output is available, the table reports number of bugs with CORRECT fixes, PRECISION, and RECALL separately for fixes ranked in ANY POSITION, in the TOP-10 POSITIONS, and in the FIRST POSITION. The rightmost column UNIQUE lists the number of distinct bugs that *only* the tool can fix correctly. Question marks represent data not available for a tool.

APR TOOL	VALID	ANY POSITION			TOP-10 POSITIONS			FIRST POSITION			UNIQUE
		CORRECT	PRECISION	RECALL	CORRECT	PRECISION	RECALL	CORRECT	PRECISION	RECALL	
DEFECTS4J											
JAID	94	35	37.2%	8.9%	25	26.6%	6.3%	14	14.9%	3.5%	11
ACS	23	18	78.3%	4.6%	18	78.3%	4.6%	18	78.3%	4.6%	11
CapGen	25	22	88.0%	5.6%	22	88.0%	5.6%	21	84.0%	5.3%	3
Elixir	41	26	63.4%	6.6%	26	63.4%	6.6%	26	63.4%	6.6%	?
HDA	?	23	?	5.8%	23	?	5.8%	13	?	3.3%	3
jGenProg	27	5	18.5%	1.3%	5	18.5%	1.3%	5	18.5%	1.3%	1
jKali	22	1	4.5%	0.3%	1	4.5%	0.3%	1	4.5%	0.3%	0
Nopol	35	5	14.3%	1.3%	5	14.3%	1.3%	5	14.3%	1.3%	2
SimFix	56	34	60.7%	8.6%	34	60.7%	8.6%	34	60.7%	8.6%	12
SketchFix	26	19	73.1%	4.8%	?	?	?	9	34.6%	2.3%	0
SketchFixPP	?	34	?	8.6%	?	?	?	?	?	?	3
ssFix	60	20	33.3%	5.1%	20	33.3%	5.1%	20	33.3%	5.1%	1
xPar	?	4	?	1.0%	4	?	1.0%	?	?	?	?
INTROCLASSJAVA											
JAID	84	69	82.1%	26.7%	43	51.2%	16.7%	30	35.7%	11.6%	?
CapGen	?	25	?	9.7%	?	?	?	?	?	?	?
JFix	?	19	?	7.4%	?	?	?	?	?	?	?
S3	?	22	?	8.5%	?	?	?	?	?	?	?
QUIXBUGS											
JAID	11	9	81.8%	22.5%	9	81.8%	22.5%	4	36.4%	10.0%	?
Astor	11	6	54.5%	15.0%	?	?	?	?	?	?	?
Nopol	4	1	25.0%	2.5%	?	?	?	?	?	?	?

AE [37], and TrpAutoRepair [38] produced [14] valid fixes for 37%, 20%, and 32% of the bugs in the original IntroClass C benchmark; the experiments [14] do not analyze how many of these fixes are *correct*.

JAID produced correct fixes for more bugs than any other automated repair tools for Java—and fixed 11 bugs in DEFECTS4J that no other tools can fix. JAID is less precise than some other tools on DEFECTS4J bugs, but it is more precise on INTROCLASSJAVA and QUIXBUGS bugs.

TABLE 11: Templates used more commonly by JAID to produce *correct* fixes. For bugs in each benchmark, the table lists the numbers of correct fixes produced by JAID using a semantic action (s), a mutation action (m), or a control-flow action (c); and the number of correct fixes that use one of the five schemas A–E in Fig. 2. Since JAID may produce multiple correct fixes for the same bug, the total number of fix actions or fix schemas listed is greater than the number of bugs correctly fixed by JAID.

BENCHMARK	FIX ACTION			FIX SCHEMA				
	s	m	c	A	B	C	D	E
DEFECTS4J	14	17	13	5	15	4	6	18
INTROCLASSJAVA	71	42	2	13	36	6	24	42
QUIXBUGS	4	3	2	2	4	1	0	3
OVERALL	89	62	17	20	55	11	30	63

4.4 RQ4: Templates and Heuristics

Templates. As explained in Sec. 2.3, JAID uses three kinds of *fix actions* as templates to build patches: *semantic* actions, *mutation* actions, and *control-flow* actions. Tab. 11 shows how often each kind of fix action was used to produce *correct* fixes. Semantic and mutation actions are the most frequently used fix action kinds, but control-flow actions are also needed for a significant fraction of correct fixes. This indicates that the three kinds of actions are often complementary and all contribute to JAID's effectiveness.

JAID patches a buggy program by injecting fix actions using one of the five different *fix schemas* in Fig. 2. A fix schema may execute the fix action unconditionally (schemas A and E) or guarded by an **`if`** condition (schemas B, C, and D); and may execute it instead of an existing statement (schemas D and E) or in addition to it (schemas A, B, and C). Tab. 11 indicates that schemas B and E are used a bit more

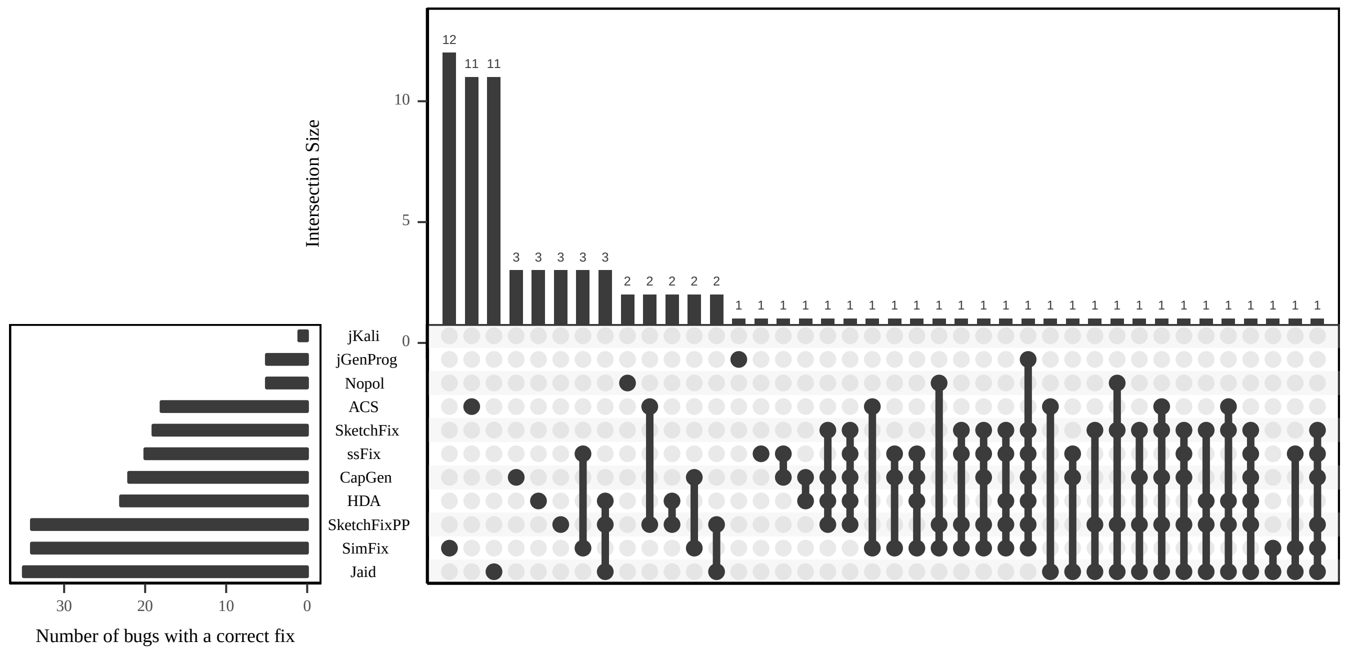


Fig. 8: Partitioning of DEFECTS4J bugs according to which tool can fix them. Each vertical bar measures the number of bugs that a certain combination of tools (indicated by connected dots in the lower part of the diagram) can correctly fix that no other tools can. For example, the leftmost column indicates that SimFix can correctly fix 12 bugs that no other tool can; the eighth column from the left indicates that HDA, SketchFixPP, and JAID can all fix the same 3 bugs that no other tools can fix. The horizontal bars on the left report how many bugs in total each tool can fix.

often, but there is no schema that is overwhelmingly more useful, and all are required to be able to fix a wide choice of bugs.

Number of snapshots. JAID builds a variable number of *snapshots* to abstract program state based on its heuristics; in its default settings (Sec. 3.5), it uses up to 1500 snapshots to drive the rest of the fixing process. It's clear that the number and variety of snapshots may affect the effectiveness (more snapshots means more precise abstractions) and efficiency (more snapshots means longer analysis times) of the fixing process. To understand the trade-off in a quantitative way, we ran a series of experiments where JAID used a different number of snapshots to fix all 693 bugs in the three benchmarks DEFECTS4J, INTROCLASSJAVA, and QUIXBUGS. Fig. 9 plots various measures of JAID's performance (number of valid and correct fixes, overall precision and recall, running time per bug, and running time until a valid or correct fix is found) as a function of the number of used snapshots—from 100 to 1500 in 100-snapshot increments. Overall, the data confirms the intuition that more snapshots means longer running times (Fig. 9c and Fig. 9d) but also a greater chance of success (Fig. 9a) and better precision (Fig. 9b). However, while the running time grows uniformly proportionally to the number of snapshots, there are diminishing returns in adding more snapshots to increase precision and recall, since as the search space becomes larger it is harder to search it effectively. Thus, further substantial progress in effectiveness is not only a matter of extending the search space but requires more structured ways of exploring it.

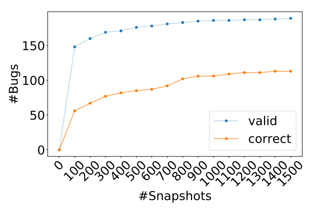
TABLE 12: How JAID's effectiveness changes using different fault localization techniques. For each of the five spectrum-based Fault Localization algorithms of Tab. 5, the table reports the number of fixes in each benchmark that JAID could repair with a Valid or a Correct fix in ALL benchmarks, and in each benchmark DEFECTS4J, INTROCLASSJAVA, and QUIXBUGS individually. The first row refers to JAID using its original fault-localization technique (as in the rest of the experimental evaluation).

	ALL		DEFECTS4J		INTROCLASSJAVA		QUIXBUGS	
FL	c	v	c	v	c	v	c	v
JAID	113	189	35	94	69	84	9	11
Barinel	111	191	33	96	69	84	9	11
DStar	111	191	33	96	69	84	9	11
Ochiai	111	191	33	96	69	84	9	11
Op2	111	191	33	96	69	84	9	11
Tarantula	111	191	33	96	69	84	9	11

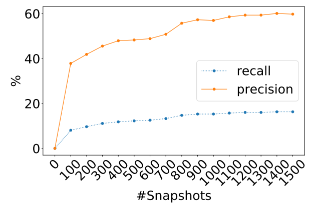
JAID's templates are all needed to be able to effectively fix bugs with different characteristics. Reducing the number of snapshots used by JAID improves its running time but also reduces the number of bugs that can be fixed.

4.5 RQ5: Fault Localization

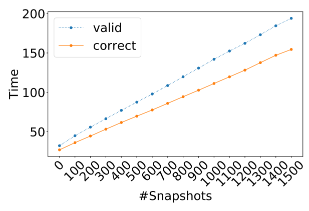
Do JAID's effectiveness and efficiency depend on the fault-localization algorithm it uses? To answer this question, we ran five different variants of JAID, each using one of the fault localization algorithms of Tab. 5 instead of JAID's default algorithm (used in the rest of the experiments). Tab. 12



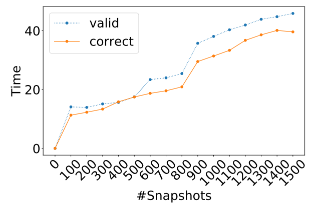
(a) Number of bugs with valid or correct fixes.



(b) Precision and recall.



(c) Average running time per bug.



(d) Average time until a valid or correct fix is found.

Fig. 9: How effectiveness and efficiency of JAID depend on the number of snapshots used for fixing. Each figure plots the number of bugs with a valid or correct fix (Fig. 9a), overall precision and recall (Fig. 9b), average running time per bug (Fig. 9c), and average time until a valid or a correct fix is built (Fig. 9d) as a function of the number of snapshots used for fixing. The data comes from experiments using all 693 bugs in the three benchmarks DEFECTS4J, INTROCLASSJAVA, and QUIXBUGS.

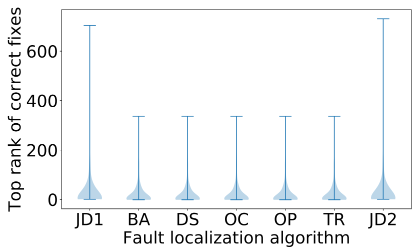


Fig. 10: Violin plots of the top rank of correct fixes for all bugs fixed by JAID using different fault localization algorithms: JAID's custom algorithm (JD), Barinel (BA), DStar (DS), Ochiai (OC), Op2 (OP), and Tarantula (TR).

shows the number of faults fixed (with a correct fix, or just with a valid fix) in each case. The differences between fault-localization techniques are very small: there are only three bugs in total that JAID can or cannot fix depending on the fault localization algorithm it uses. Precisely, JAID failed to produce any valid fixes for bug `Lang53` in DEFECTS4J when using its default fault localization algorithm, since the one snapshot from which valid fixes could be derived was not ranked among the top 1500 most suspicious; in contrast, JAID could only correctly fix bugs `Closure33` and `Chart9` in DEFECTS4J using its default fault localization algorithm, since the other five algorithms ranked the "useful" snapshots too low to be used.

Fig. 10 looks at how the *top rank* of a correct fix varies with the choice of fault localization algorithm. Here too differences between JAID's default algorithm and any other algorithm are quite limited—with JAID's default algorithm leading to a greater variance in the rank but no noticeable differences on average.

Finally, Fig. 11 plots JAID's running time using different fault localization algorithms. Once again, there are no marked differences between JAID's default algorithm and any other algorithm.

In summary, JAID's overall technique is remarkably robust with respect to the choice of fault localization algorithm. Note, however, that all fault localization algorithms are *spectrum-based*, and hence process the same information (the spectrum over the available executions) using slightly different heuristic formulas (see Tab. 5). Using radically different fault localization approaches [24], [39] may still have a significant impact on the performance of automated program repair.

JAID's effectiveness and efficiency are largely independent of the spectrum-based fault localization algorithm that is used.

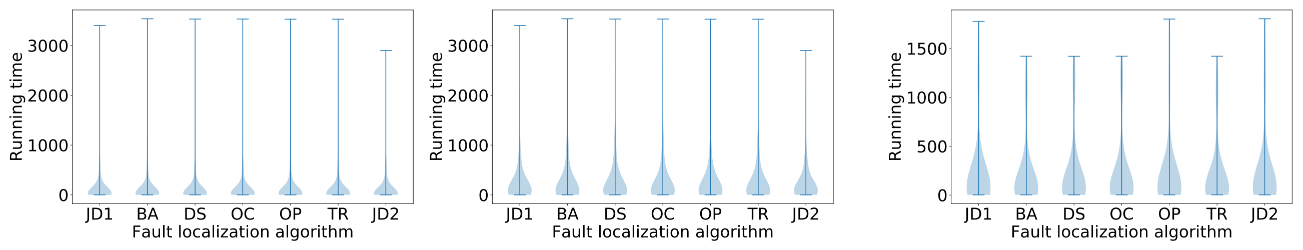
5 THREATS TO VALIDITY

Construct validity indicates whether the measures used in the experiments are suitable. We classify a fix as *correct* if it is semantically equivalent to a programmer-written fix. Since we assess semantic equivalence manually, different programmers may provide different assessments; to mitigate this threat, we were conservative in evaluating equivalence—if a fix does not clearly produce the same behavior as the fix in DEFECTS4J or QUIXBUGS for the same bug, we classify it as incorrect. This approach is consistent with what was done by other researchers, and seems to be reasonably reliable as a way of assessing correctness [40].

We measured, and compared, precision and recall relative to *all bugs* from the three benchmarks, although most APR techniques only run experiments on a subset of the DEFECTS4J bugs whose features have a chance of being fixable. Using the largest possible denominator ensures that measures are comparable between different tools, and is consistent with the ultimate ambition of developing APR techniques that are as widely applicable as possible.

Tools, and their experimental evaluations, often differ in how they deal with multiple valid fixes for the same bugs. In the tool comparison, we counted all correct fixes generated by each tool that were reported in the experiments, and we reported separate measures of precision according to how many valid fixes are inspected. This gives a nuanced picture of the results, which must however be taken—as usual—with a grain of salt: different tools may focus on achieving a better ranking vs. correctly fixing more bugs, and we do not imply that there is one universal measure of effectiveness. Anyway, our evaluation is widely applicable—including to papers that may not detail this aspect—and is in line with what was done in other evaluations [1], [3], [34]–[36].

Internal validity indicates whether the experimental results soundly support the findings. Comparing the



(a) All bugs. (b) Bugs with a valid fix. (c) Bugs with a correct fix.

Fig. 11: Violin plots of the overall running time of JAID using different fault localization algorithms: JAID's custom algorithm (JD), Barinel (BA), DStar (DS), Ochiai (OC), Op2 (OP), and Tarantula (TR).

performance—running time, in particular—of different APR techniques is a particularly delicate matter because of a number of confounding factors. First of all, the experiments should all run on the same hardware and runtime environment, using comparable configurations (e.g., in terms of timeouts). Techniques using randomization, such as jGenProg, require several repeated runs to get quantitative results that are representative of a typical run [41]. Some techniques, such as ACS, HDA, ssFix, and CapGen, rely on a time-consuming preprocessing stage that mines code repositories (and is crucial for effectiveness), and hence it is unclear how to appropriately compare them to techniques, such as JAID, that do not depend on this auxiliary information. Fault localization is also an explicit input to most other APR techniques. In all, we used standard, clearly specified settings for the experiments with JAID, and we relied on the overall results—in terms of correct fixes—reported in other tools' experiments. In contrast, we refrained from qualitatively compare tools in measures of performance, which depend more sensitively on having a controlled experimental setup, and which we therefore leave to future work.

External validity indicates whether the experimental findings generalize. Our experimental evaluation targets a large and varied selection of bugs in three benchmarks. The DEFECTS4J dataset is a varied collection of bugs, carefully designed and maintained to support realistic and sound comparisons of the effectiveness of all sorts of analyses based on testing and test-case generation; it has also become a *de facto* standard to evaluate APR techniques for Java. The INTROCLASSJAVA dataset contains bugs representative of mistakes commonly made by novice programmers in writing small programs. Bugs in the QUIXBUGS dataset are representative of mistakes programmers often make in implementing algorithms that are challenging to get right. Overall, using bugs with distinct characteristics from different sources helps to mitigate the risk that our experiments overfit the subjects. As future work, we plan to run JAID on other open-source Java projects; we see no intrinsic limitations that would prevent JAID from working reliably on other projects as well.

6 RELATED WORK

Automated program repair has become a bustling research area in the course of just a few years. The first APR techniques [42], [43] used genetic algorithms to search the space of possible fixes for a valid one. GenProg [43] pioneered the "generate-and-validate" approach, where many plausible fixes are generated based on heuristics, and then are validated against the available tests. More recently, others [9], [21], [33], [44]–[46] have pursued the "constraint-based" approach, where fixes are constructed to satisfy suitable constraints that correspond to their validity. The two approaches are not sharply distinct, in that fixes generated by constraint-based techniques may still require validation if the constraints they satisfy by construction are not sufficiently precise to ensure that they are correct—as it often happens when dealing with incomplete specifications. Nevertheless, the categorization remains useful; we devote more attention to generate-and-validate techniques, since JAID belongs to this category, and thus is more directly comparable to them. We also focus the discussion on recent techniques that are applicable to Java programs, since these are directly comparable to JAID; for an exhaustive list of APR techniques, see the annotated bibliography [47].

Generate-and-validate. GenProg [43] is based on a genetic algorithm that mutates the code of a faulty C function by deleting, adding, or replacing code taken from other portions of the codebase—following the intuition [48] that existing code is also applicable to patch incorrect functionality. PAR [49] bases the generation of fixes on ten patterns, selected based on a manual analysis of programmer-written fixes, which helps generate fixes that are more readable, and possibly easier to understand. Since PAR is not publicly available, xPar [6], [7] reimplements PAR. A complementary approach [50] suggests to use *anti*-patterns, trying to capture fixes that are likely to be incorrect but still pass validation. SketchFix [17] replaces suspicious expressions with "holes" via AST node-level code transformation and employs a sketch engine to systematically synthesize expressions with predefined structures *on demand* to fill the holes. By integrating fix generation and validation, SketchFix significantly outperforms existing G&V techniques in proposing *expression-level* fixes.

A performance comparable to that of GenProg can be achieved using a simplistic APR tool Kali [1], which repairs faulty C programs by just removing or skipping code; Martinez and Monperrus [2] developed jKali, a Java implementation of Kali.

By also relying on contracts (specifications embedded in the program text) AutoFix [4], [36], [51] was the first general-purpose APR technique able to produce a significant number of correct fixes. One of the ideas behind JAID's design is to generalize AutoFix's state-based analysis to work on Java code without contracts, so as to improve the quality of the generated fixes without sacrificing applicability.

Code mining. A number of recent approaches are based on preprocessing a large dataset of programmer-written code snippets to learn useful features of effective repairs. SearchRepair [52] encodes program behavior as input/output relational constraints, and then generates fixes by searching the dataset for snippets that capture the desired input/output behavior. HDA [6] also leverages a model of programmer-written code built by mining software repositories, but combines it with a mutation-based syntax-driven analysis similar to GenProg: mutants that are "more similar" to what the learned model prescribes are preferred in the search for a repair. Elixir [19] learns a logistic regression model from existing patches and uses the model to identify promising fixes to faulty method calls. ssFix [23] leverages existing code that is syntactically related to the fixing context to produce patches. CapGen [16] exploits fault context information to prioritize expression-level fixes so that fixes more likely to be correct are generated first. SimFix [22] utilizes both existing patches and similar code to repair faults. The idea of mining programmer-written code is applicable to other APR approaches, including JAID, as a way to provide additional information that reduces the chance of overfitting.

Condition synthesis. Constraint-based approaches [44], [46] often target the synthesis of *conditions* in if statements or loops, since changing those conditions often affects the control flow in decisive ways. For example, Angelix [33] introduced an efficient representation of constraints, combined with a symbolic execution analysis, which helps it scale.

Nopol [9] only targets conditional expressions, and uses a form of angelic debugging [53], [54] to reconstruct the expected value of a condition in passing vs. failing runs; based on it, it synthesizes a new conditional expression using an SMT solver. SPR [55] and Prophet [34] also combine condition synthesis with a dynamic analysis of the value each abstract conditional expression should take to make all tests pass, which helps aggressively prune the search space when no plausible repair exists. ACS [7] significantly improves the precision of condition synthesis based on a combination of data- and control-dependency analysis, and mining API documentation and Boolean predicates in existing projects.

7 Conclusions and Future Work

We presented an extensive experimental evaluation of JAID, a technique and tool for the automated repair of Java programs. In experiments involving all 693 bugs from the DEFECTS4J, INTROCLASSJAVA, and QUIXBUGS benchmark suites, JAID produced correct fixes for 113 bugs with a precision of nearly 60%. At the time of writing, 11 of these bugs could not be fixed by any other state-of-the-art automated repair technique for Java.

Like most techniques based on dynamic analysis and generate-then-validate fix generation, JAID's running times are often reasonable (on average, 24 minutes to generate a correct fix) but can become very long on a few "hard" bugs. We plan to pursue two directions to improve JAID's running time performance: 1) fine-grained test selection strategies could curtail validation time by only rerunning the tests whose behavior may have changed; 2) parallelizing the execution of the validation phase should be straightforward, since different unit tests are obviously independent.

Major progress in the precision and applicability of automated program repair typically requires tapping into additional sources of information about program behavior: JAID relies on a rich state-based dynamic analysis; other approaches mine repositories of code and fixes [6], [7]. While our experiments suggest that JAID's overall effectiveness does not depend much on the details of its spectrum-based fault localization algorithm, they also indicate that further substantial progress would probably require to step up the precision of fault localization in a way that it can incorporate such additional sources of information. To this end, we plan to integrate fault localization closely with JAID's overall repair process.

Acknowledgments

This work was partially supported by the Hong Kong RGC General Research Fund (GRF) under grant PolyU 152703/16E, by the Hong Kong Polytechnic University under internal fund 1-ZVJ1, and by the Swiss National Science Foundation (SNF) under grant Hi-Fi 200021-182060.

References

[1] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2015, pp. 24–36.

[2] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset," *Empirical Software Engineering*, 2016.

[3] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? Overfitting in automated program repair," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 532–543.

[4] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2010, pp. 61–72.

[5] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, 2017, pp. 637–647.

[6] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 213–224.

[7] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. ACM, Aug. 2017.

[8] W. E. Wong, V. Debroy, and B. Choi, "A family of code coverage-based heuristics for effective fault localization," *Journal of Systems and Software*, vol. 83, no. 2, pp. 188–208, 2010.

[9] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. R. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in Java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2017.

[10] T. Durieux and M. Monperrus, "IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs," Universite Lille 1, Tech. Rep., 2016.

[11] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "QuixBugs: a multi-lingual program repair benchmark set based on the Quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. ACM, 2017, pp. 55–56.

[12] T. Durieux, F. M. Delfim, M. Martinez, and R. Abreu, "Empirical review of Java program repair tools: a large-scale experiment on 2,141 bugs and 23,551 repair attempts," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, (ESEC/SIGSOFT FSE)*. ACM, 2019, pp. 302–313.

[13] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. L. Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 102–113.

[14] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. T. Devanbu, S. Forrest, and W. Weimer, "The ManyBugs and IntroClass benchmarks for automated repair of C programs," *IEEE Trans. Software Eng.*, vol. 41, no. 12, pp. 1236–1256, 2015.

[15] X. D. Le, D. Chu, D. Lo, C. Le Goues, and W. Visser, "JFIX: semantics-based repair of java programs via symbolic pathfinder," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, 2017, pp. 376–379.

[16] M. Wen, J. Chen, R. Wu, D. Hao, and S. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 1–11.

[17] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 12–23.

[18] M. Martinez and M. Monperrus, "ASTOR: A program repair library for Java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA) – Demonstrations Track*. ACM, 2016, pp. 441–444.

[19] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "ELIXIR: effective object oriented program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, 2017, pp. 648–659.

[20] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A comprehensive study of automatic program repair on the QuixBugs benchmark," https://arxiv.org/abs/1805.03454, 2019.

[21] X. D. Le, D. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax- and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 593–604.

[22] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, 2018, pp. 298–309.

[23] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, 2017, pp. 660–670.

[24] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Software Eng.*, vol. 42, no. 8, pp. 707–740, 2016.

[25] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 609–620.

[26] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 273–282.

[27] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, Nov. 2009.

[28] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 3, pp. 11:1–11:32, Aug. 2011.

[29] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, "Spectrum-based multiple fault localization," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 88–99.

[30] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2014.

[31] S. Wang, M. Wen, X. Mao, and D. Yang, "Attention please: Consider mockito when evaluating newly proposed automated program repair techniques," in *Proceedings of the Evaluation and Assessment on Software Engineering*, ser. EASE '19. New York, NY, USA: ACM, 2019, pp. 260–266.

[32] V. Sobreira, T. Durieux, F. M. Delfim, M. Monperrus, and M. de Almeida Maia, "Dissection of a bug dataset: Anatomy of 395 patches from defects4j," in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, 2018, pp. 130–140.

[33] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 691–701.

[34] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2016, pp. 298–312.

[35] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 3–13.

[36] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," *IEEE Transactions on Software Engineering*, vol. 40, no. 5, pp. 427–449, 2014.

[37] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 356–366.

[38] Y. Qi, X. Mao, and Y. Lei, "Efficient automated program repair through fault-recorded testing prioritization," in *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*, 2013, pp. 180–189.

[39] T. Xu, L. Chen, Y. Pei, T. Zhang, M. Pan, and C. A. Furia, "Restore: Retrospective fault localization enhancing automated program repair," https://arxiv.org/abs/1906.01778, 2019.

[40] X. D. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. S. Pasareanu, "On reliability of patch correctness assessment," in *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. IEEE / ACM, 2019, pp. 524–535.

[41] A. Arcuri and L. C. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Softw. Test., Verif. Reliab.*, vol. 24, no. 3, pp. 219–250, 2014.

[42] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2008, pp. 162–168.

[43] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *31st International Conference on Software Engineering (ICSE)*. IEEE, 2009, pp. 364–374.

[44] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: program repair via semantic analysis," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.

[45] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus, "Automatic repair of buggy if conditions and missing preconditions with SMT," in *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*. ACM, 2014, pp. 30–39.

[46] S. Mechtaev, J. Yi, and A. Roychoudhury, "DirectFix: Looking for simple program repairs," in *37th International Conference on Software Engineering (ICSE)*, vol. 1. IEEE, 2015, pp. 448–458.

[47] M. Monperrus, "Automatic software repair: a bibliography," University of Lille, Tech. Rep. hal-01206501, 2015.

[48] M. Martinez, W. Weimer, and M. Monperrus, "Do the fix ingredients already exist? An empirical inquiry into the redundancy

assumptions of program repair approaches," in *36th International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 492–495.

[49] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 802–811.

[50] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2016, pp. 727–738.

[51] Y. Pei, Y. Wei, C. A. Furia, M. Nordio, and B. Meyer, "Code-based automated program fixing," in *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2011, pp. 392–395.

[52] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search," in *30th IEEE/ACM International Conference on Automated Software Engineering, (ASE)*. IEEE, 2015, pp. 295–306.

[53] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th International Conference on Software Engineering (ICSE)*. ACM, 2006, pp. 272–281.

[54] S. Chandra, E. Torlak, S. Barman, and R. Bodík, "Angelic debugging," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. ACM, 2011, pp. 121–130.

[55] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 166–178.