Contract-Based Program Repair without the Contracts

Liushan Chen* · Yu Pei* · Carlo A. Furia†
*Department of Computing, The Hong Kong Polytechnic University {cslschen,csypei}@comp.polyu.edu.hk
†Department of Computer Science and Engineering, Chalmers University of Technology, Sweden bugcounting.net

***Abstract*—Automated program repair (APR) is a promising approach to automatically fixing software bugs. Most APR techniques use tests to drive the repair process; this makes them readily applicable to realistic code bases, but also brings the risk of generating spurious repairs that overfit the available tests. Some techniques addressed the overfitting problem by targeting code using contracts (such as pre- and postconditions), which provide additional information helpful to characterize the states of correct and faulty computations; unfortunately, mainstream programming languages do not normally include contract annotations, which severely limits the applicability of such contract-based techniques.**

This paper presents JAID, a novel APR technique for Java programs, which is capable of constructing detailed state abstractions—similar to those employed by contract-based techniques—that are derived from regular Java code without any special annotations. Grounding the repair generation and validation processes on rich state abstractions mitigates the overfitting problem, and helps extend APR's applicability: in experiments with the DEFECTS4J benchmark, a prototype implementation of JAID produced genuinely correct repairs, equivalent to those written by programmers, for 25 bugs—improving over the state of the art of comparable Java APR techniques in the number and kinds of correct fixes.

I. INTRODUCTION

Every general software analysis technique based on a finite collection of tests is prone to *overfitting* them. Automated program repair (APR) is no exception; in particular, overfitting is likely to cripple the performance of APR tools following the *generate-then-validate* paradigm that was pioneered by GenProg [33], where each heuristically generated *candidate* repair—a source code patch—undergoes testing, and only the candidates that pass all available tests for the method being repaired are classified as *valid* and returned as fix suggestions. Since validation is against a finite—often small—number of tests, there is no guarantee that a valid repair is genuinely *correct* against a complete, and implicit, specification of the method. Indeed, experiments have repeatedly confirmed [19], [28], [29] that automated program repair techniques are prone to producing a significant fraction of valid but incorrect repairs, which merely happen to pass all available tests but are clearly inadequate from a programmer's perspective.

The AutoFix technique for APR [31] mitigated the overfitting problem by using *contracts*, made of *assertions* such as pre- and postconditions, as additional information to improve the precision of repair generation and validation. Even if the contracts used by AutoFix are far from being detailed, let alone complete, method specifications, they significantly help increase the fraction of correct fixes that can be generated [25]. Unfortunately, even such simple contracts are hardly ever available in the most widely used programming languages.[1] Can we still generalize some of the techniques used for contract-based program repair to work effectively *without* user-written contracts?

In this paper we describe JAID: a technique and tool for **automated program repair of Java programs** that is based on detailed, *state-based* dynamic program analyses—akin those employed by contract-based techniques such as AutoFix, but working on regular Java code (without any contracts). State abstractions drive both the generation and the validation stages of JAID, and help construct high-quality fixes: in experiments targeting bugs from the DEFECTS4J curated collection, JAID produced repairs passing all available tests for 31 of the bugs, and *correct repairs*—equivalent to those written by programmers—for 25 of the bugs. These results are close to, or outperform, other comparable tools for the automated program repair of Java programs in terms of total number of correct repairs and precision, and include the first automatically produced correct repairs for 14 bugs of DEFECTS4J that were previously outside the capabilities of APR. JAID is also the first APR technique that achieves high levels of precision *without relying on additional input* other than tests and faulty code; in contrast, other recent high-precision APR techniques [13], [35] analyze a large number of project repositories to collect additional information that guides fixing.

This paper's key **contributions**, which bolster JAID's performance, include techniques to build a rich *abstraction of object state*. In turn, the state abstraction relies on a *purity* analysis of functions—only functions that are pure, that is without side effects, can be safely used to characterize state. Whereas techniques, such as AutoFix, that use programmer-written contracts can easily rely on the functions used as predicates in the contracts, JAID has to extract similar information from regular code without annotations. To curb the number of candidate fixes that are generated and validated, JAID relies on fault *localization* and *ranking heuristics*, which help identify program states that are likely to be implicated with faulty behavior; both fault localization and ranking are crucially informed by JAID's detailed state-based abstractions. Thanks

[1] AutoFix targets the Eiffel programming language, where contracts are embedded in the program code and routinely written by programmers.

to these techniques, JAID can generate correct fixes that are based on a more “semantic” analysis of how to modify the object state to avoid a failure—beyond just working around the existing implementation by syntactically modifying it, as most other APR tools do.

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

II. An Example of JAID in Action

Apache Commons is a widely used Java library that extends Java’s standard API with a rich collection of utilities. Class `WordUtil` of package `org.apache.commons.lang` includes a method `abbreviate` to simplify strings with spaces: given a string `str`, lower and upper indexes `lower` and `upper`, and another string `appendToEnd`, the method returns a string obtained by truncating `str` at the first index between `lower` and `upper` where a space occurs, and replacing (or abbreviating) the truncated suffix with `appendToEnd`. For example, `abbreviate("Apache␣Commons␣library", 9, 18, "+")` returns the string `"Apache␣Commons+"`.

Lst. 1 shows the implementation of `abbreviate` at commit `#cfff06bead` of the library, which is also part of DEFECTS4J’s curated collection of defects (bug Lang-45). The implementation begins by handling a number of special cases but, unfortunately, it misses the case when `lower` is greater than `str`’s length: the `index` of the first occurrence of a space from `lower` will then be `-1` (corresponding to a failing search for such a space in the call at line 16), and `upper` will be greater than or equal to `lower` (possibly after being adjusted at lines 12–13), and thus also greater than `str.length()`; in these conditions, the call to method `substring` at line 19 throws an `IndexOutOfBoundsException`.

The maintainers of Apache Commons fixed this fault in a later version of the library by resetting `lower` to `str.length()` to ensure that the case `lower > str.length()` never occurs, as shown in the patch of Lst. 2, to be inserted right before line 9 in Lst. 1. DEFECTS4J includes a test that triggers this fault in `abbreviate`, to avoid reintroducing the same mistake in future revisions of the code.

After running for about 70 minutes, JAID produces a number of fix suggestions for the fault of `abbreviate`, including the fix in Lst. 3; this fix is equivalent (nearly identical) to the programmer-written fix, and thus completely removes the source of error by handling the special case correctly. To generate fixes for `abbreviate`, JAID only needs the source code of the faulty implementation, as well as the programmer-written tests that exercise the method. DEFECTS4J actually includes only one test—the test triggering the fault—for this bug; JAID can produce a correct fix even with such limited information.

```
1  public static String abbreviate
2          (String str, int lower, int upper, String appendToEnd) {
3    if (str == null) {
4      return null;
5    }
6    if (str.length() == 0) {
7      return StringUtils.EMPTY;
8    }
9    if (upper == -1 || upper > str.length()) {
10     upper = str.length();
11   }
12   if (upper < lower) {
13     upper = lower;
14   }
15   StringBuffer result = new StringBuffer();
16   int index = StringUtils.indexOf(str, "␣", lower);
17   if (index == -1) {
18     // throws IndexOutOfBoundsException if lower > str.length()
19     result.append(str.substring(0, upper));
20     if (upper != str.length()) {
21       result.append(StringUtils.defaultString(appendToEnd));
22     }
23   } else if (index > upper) {
24     result.append(str.substring(0, upper));
25     result.append(StringUtils.defaultString(appendToEnd));
26   } else {
27     result.append(str.substring(0, index));
28     result.append(StringUtils.defaultString(appendToEnd));
29   }
30   return result.toString();
31 }
```

Listing 1. Faulty method `abbreviate` from class `StringUtils` in package `org.apache.commons.lang`.

```
8a10,12
> if (lower > str.length()) {
>   lower = str.length();
> }
```

Listing 2. Programmer-written fix to the fault in `abbreviate`.

```
8a10,12
> if (lower >= str.length()) {
>   lower = str.length();
> }
```

Listing 3. JAID’s correct fix to the fault in `abbreviate`.

To our knowledge, JAID is the first APR tool that can correctly repair the fault of `abbreviate`; no other existing tools even provided so-called test-suite adequate fixes, which spuriously pass all available tests avoiding the failure, but do not correctly fix the behavior in the same way that the developers did. Key to JAID’s success is its capability of constructing rich state-based abstractions of a program’s behavior, which improves the accuracy of fault localization and guides the creation of state-modifying fixes in response to failing conditions.

III. 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. 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. III-A. Fault localization (Sec. III-B) identifies states and locations (snapshots) that are suspect of being implicated in the failure under repair. Fix generation (Sec. III-C and Sec. III-D) builds code snippets that avoid reaching such suspicious states and locations by modifying the program state, the control flow, or by other simple heuristics. Generated fixes are validated against the available tests (Sec. III-E); the fixes that pass validation are presented to the user, heuristically ranked according to how likely they are correct (Sec. III-F).

The rest of this section describes how JAID repairs 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.

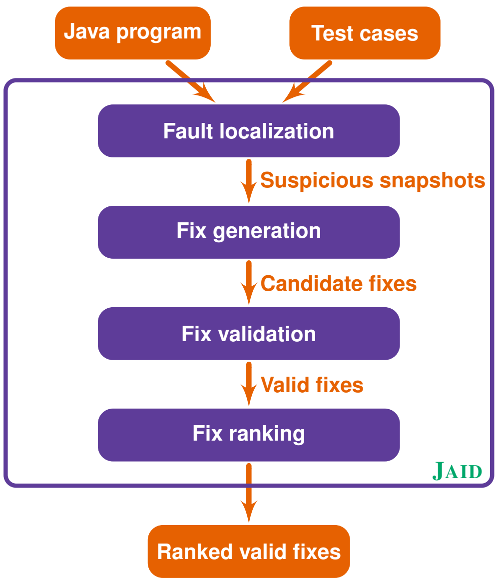

Fig. 1. An overview of how JAID works. Given a Java program and a set of test cases, including at least one failing test, JAID identifies a number of suspicious snapshots, each indicating a location and an abstraction of the program state at that location that may be implicated in the failure; based 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.

A. 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, the 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 selects the expressions in M_ℓ as follows.

Expressions. A type is *monitorable* if it is a reference type or a primitive type (numeric types such as **`int`**, and **`boolean`**). E_ℓ denotes the set of all *basic* expressions of monitorable types at ℓ, namely 1) local variables (including `fixMe`'s arguments) declared inside `fixMe` that are visible at ℓ; 2) attributes of class `FC` that are visible at ℓ; 3) expressions anywhere inside `fixMe` that can be evaluated at ℓ (that is, they only involve items visible at ℓ), and that don't obviously have side effects (namely, we exclude assignments used as expressions, self increment and decrement expressions, and creation expressions using **`new`**). X_ℓ denotes the set of all *extended* expressions of monitorable types at ℓ: for each basic expression of reference type $r \in E_\ell$, X_ℓ includes: 1) r`.f()` for every argumentless function `f` of the class corresponding to r's type that returns a monitorable type and is callable at ℓ; 2) only if r is **`this`**, r`.a`, for every attribute `a` of the class corresponding to r's type that is readable at ℓ.

For example, the extended expressions X_9 at line 9 in method `abbreviate` of Lst. 1 include `lower` (an argument of `abbreviate`), `str.length()` (a call of function `length()` on `abbreviate`'s argument `str`), and `upper < lower` and `str == null` (both appearing in `abbreviate`).

Purity analysis. One lesson that we can draw from the experience of contract-based APR [25] is that constructing a rich set of expressions that abstract the program state can help support more accurate fault localization and fix generation, and ultimately the construction of higher-quality "semantic" fixes that are less prone to overfitting. However, monitoring a rich set of expressions extracted from the program text does not work as well in languages such as Java as it does in languages that support contracts. In the latter, programmers specifically equip classes with public query methods that are *pure*—they are functions that return a value without changing the state of their target objects—and can be used in the contracts to characterize the program state in response to method calls; these methods are thus easily identifiable and natural candidates to construct state abstractions reliably. In Java, in contrast, programmers need not follow such a discipline of separating pure functions from state-changing procedures, and methods that return a value but have side effects are indeed common. Clearly, a function that is not pure is unsuitable for abstracting and monitoring an object's state.

To identify which expressions can reliably be used for state monitoring, JAID performs a dynamic purity analysis on all expressions that include method invocations. Given an expression r of reference type, the set W_r of r's *watch expressions* consists of: 1) all subexpressions S_r of r that do not include method invocations; 2) for each subexpression $s \in S_r$, s`.a` for every attribute `a` of the class corresponding to s's type. Note that watch expressions are constructed so that they are syntactically free from side effects.

An expression r of reference type is then considered *pure* if evaluating it does not alter the value of its watch expressions. Precisely, at every location ℓ in the method `fixMe` under repair, 1) first, JAID records the value $\sigma = \langle \sigma_1, \ldots, \sigma_m \rangle$ of all watch expressions, where σ_k is the value of $w_k \in W_r$, for $1 \leq k \leq m$, before evaluating r; 2) then, JAID evaluates r; 3) finally, JAID records again the value $\sigma' = \langle \sigma'_1, \ldots, \sigma'_m \rangle$ of all watch expressions, where σ'_k is the value of $w_k \in W_r$, for $1 \leq k \leq m$, after evaluating r. If $\sigma = \sigma'$ at every ℓ in every test exercising `fixMe`, we call r *pure*.

State monitoring. JAID collects in M_ℓ all extended expressions in X_ℓ that are pure according to this analysis.

B. Fault Localization

The goal of fault localization is to identify *suspicious snapshots* indicating locations and states that are likely to be implicated with 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.

Boolean abstractions. The set B_ℓ includes all Boolean expressions that may appear in a snapshot at ℓ; it is constructed by combining the monitored expressions M_ℓ to create Boolean expressions as follows: 1) for each pair $m_1, m_2 \in M_\ell$ of expressions of the same type, B_ℓ includes m_1 == m_2 and m_1 != m_2; 2) for each pair $k_1, k_2 \in M_\ell$ of expressions of integer type, B_ℓ includes $k_1 \bowtie k_2$, for $\bowtie \in \{$<, <=, >=, >$\}$; 3) for each expression $b \in M_\ell$ of Boolean type, B_ℓ includes b and !b; 4) for each pair $b_1, b_2 \in M_\ell$ of expressions of Boolean type, B_ℓ includes b_1 && b_2 and b_1 || b_2.

Continuing the example of method `abbreviate` in Lst. 1, B_9 includes expressions such as `lower >= str.length()` and `!(str == null)`.

Snapshot suspiciousness. JAID computes the *suspiciousness* of every snapshot $s = \langle \ell, b, ?\rangle$ based on Wong at al.'s fault localization techniques [34]. The basic idea is that the suspiciousness of s 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 $2/(ed_s^{-1} + dy_s^{-1})$ is the harmonic mean of these two sources, but the dynamic analysis has the biggest 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.

This approach is similar to AutoFix's [25, Sec. 4.2]—which is also based on [34]—but conspicuously excludes information about the distance between ℓ and the location of failure on the control flow graph of the faulty method. AutoFix identifies failures as contract violations, which tend to be happen closer to where the program state becomes corrupted; by contrast, in JAID's setting—using tests without contracts in Java—failures normally happen when evaluating an assert statement inside a test method, and thus the distance to the location of failure within the faulty method is immaterial, and hardly a reliable indication of suspiciousness.

In the running example of method `abbreviate` in Lst. 1, the snapshot $\langle 9,$ `lower >= str.length()`, **`true`**$\rangle$ receives a high suspiciousness score because `low` and `str.length()` appear prominently in the statements around line 9, and, most important, `lower >= str.length()` holds in all failing and in no passing tests.

C. Fix Generation: Fix Actions

A snapshot $s = \langle \ell, b, ?\rangle$ 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 four kinds of *fix actions*: 1) modify the state directly by assignment; 2) affect the state that is used in an expression; 3) mutate a statement; 4) redirect the control flow. Each fix action is a (possibly compound) statement that can replace the statement at ℓ. Actions of kinds 1 and 2 are semantic—they directly target the program state; actions of kind 3 are syntactic—they tinker with existing code expressions according to simple heuristics; actions of kind 4 are the simplest—they are independent of the snapshot's information. We outline how JAID builds fix actions in the following paragraphs, based on a definition of derived expressions. Sec. IV discusses which fix actions were the most effective in the experimental evaluation.

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(···), 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 $e = \delta$ for each $\delta \in \Delta'_{\ell,b}$ whose type is compatible with e's.

In the running example of method `abbreviate` in Lst. 1, JAID includes the assignment `lower = str.length()` among the fix actions that modify the state at line 9.

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 [13], [36], 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(a1, ..., an)`, 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 **`continue`**.

D. Fix Generation: Candidate Fixes

Each fix action—built by JAID as described in the previous section—is a statement that modifies the program behavior at location ℓ in a way that avoids the state implicated by some suspicious snapshot $s = \langle \ell, b, ? \rangle$. In most cases, a fix action should not be injected into the program under repair *unconditionally*, but only when state b is actually reached during a computation. A conditional execution would leave program behavior unchanged in most cases, and only address the failing behavior when it is about to happen.

To implement such conditional change of behavior, JAID uses the *schemas* in Fig. 2 to insert fix actions into the method `fixMe` under repair at location ℓ. First, JAID instantiates every applicable schema with each fix action `action`; in addition to the fix action, schemas include the statement `oldStatement` at location ℓ in the faulty `fixMe`, and the condition `suspicious`, which is b == ? as determined by the snapshot's abstract state. Then, JAID builds *fix candidates* by *replacing* the statement at ℓ in `fixMe` by each instantiated schema.[2]

Continuing the running example of method `abbreviate` in Lst. 1, one of the fix candidates consists of the fix action `lower = str.length()` instantiating schema B: the action is executed only if `lower >= str.length()` (from the suspicious snapshot), whereas the existing statement at line 9, as well as the rest of method `fixMe`, is unchanged by the fix.

Two of the five schemas currently used by JAID to build fix candidates inject the fix action unconditionally. On the other hand, different fix actions may determine semantically equivalent fixes when instantiated. JAID performs a lightweight redundancy elimination, based on simple syntactic rules such

[2]Since each fix generated by JAID combines one fix action and one schema, it adds at most 5 new lines of codes to a patched method.

```
action;
oldStatement;
```

Listing 4. Schema A

```
if (suspicious) {
  action;
}
oldStatement;
```

Listing 5. Schema B

```
if (!suspicious) {
  oldStatement;
}
```

Listing 6. Schema C

```
if (suspicious) {
  action;
} else {
  oldStatement;
}
```

Listing 7. Schema D

```
// oldStatement
action;
```

Listing 8. Schema E

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

```
// class being repaired
class FC {
  U fixMe_(T1 a1, T2 a2, ...) throws IllegalStateException {
    switch (Session.getActiveFixId()) {
      case 0: return fixMe(a1, a2, ...);    // call faulty method
      case 1: return fixMe_1(a1, a2, ...);  // call fix candidate 1
      .
      .
      .
      case n: return fixMe_n(a1, a2, ...);  // call fix candidate n
      default: throw new IllegalStateException();
    }
  }
}
```

Listing 9. How multiple fix candidates are woven into a single class.

as that `x == y` is equivalent to `!(x != y)`. In future work, we plan to introduce a more aggressive redundancy elimination, for example as done in related work [32].

E. 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 T that exercise the faulty method `fixMe` against each generated candidate fix. Candidate fixes that pass all tests T are classified as *valid* (also "test-suite adequate" [19]) and retained; other candidates, which fail some tests, are discarded—as they do not fix the fault, they introduce a regression, or both.

In the example of method `abbreviate` in Lst. 1, the fix candidates **`if`** `(lower >= str.length) lower = str.length()` passes validation, since it fixes the fault and introduces no regression error.

Since JAID commonly generates a large number of candidate fixes for each fault, validation can take up a very large time spent compiling and executing tests, which may ultimately impair the scalability of JAID's APR. To curtail the time spent compiling, JAID deploys a simple form of dependency injection. All candidate fixes for a method `fixMe` become members of `fixMe`'s enclosing class `FC`: candidate fix number `k` becomes a method `fixMe_k` with signature the same as `fixMe`'s. Then, as shown in Lst. 9, a method `fixMe_`—also with the same signature—dispatches calls to any of the candidate fixes based on the value returned by static method `getActiveFixId()` of class `Session`, which supplies the dependency. This scheme only requires one compilation per method under repair, thus significantly cutting down validation time.

F. Fix Ranking

Like most 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 addresses 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.

When the ranking heuristics works, the user only inspects few top-ranked fixes to assess their correctness and whether they can be deployed into the codebase. The experimental evaluation in Sec. IV comments on the effectiveness of JAID's ranking heuristics.

IV. EXPERIMENTAL EVALUATION

We evaluate the effectiveness of JAID on DEFECTS4J [9], a large curated collection of faults and programmer-written fixes from real-world Java projects. This choice also enables us to quantitatively compare the results of JAID's evaluation to most state-of-the-art tools for APR of Java programs—which have also targeted DEFECTS4J in their experiments.

TABLE I

The bugs in DEFECTS4J: for each PROJECT in DEFECTS4J, how many thousands of lines of code (KLOC) and tests (TESTS) it includes, how many BUGS it includes in total, and how many were SELECTED for our experiments.

PROJECT		KLOC	TESTS	BUGS	SELECTED
Chart	JFreechart	96	2205	26	12
Closure	Closure Compiler	90	7927	133	56
Lang	Apache Commons-Lang	22	2245	65	22
Math	Apache Commons-Math	85	3602	106	42
Time	Joda-Time	27	4130	27	6
TOTAL		320	20109	357	138

A. Subjects

Our experiments target revision `#a910322b` of DEFECTS4J, which includes 357 bugs in 5 projects: Chart (26 bugs), Closure (133 bugs), Lang (65 bugs), Math (106 bugs), and Time (27 bugs). Each bug in DEFECTS4J has a unique identifier, corresponds to two versions—buggy and fixed (by a programmer)—of the code (which may span multiple methods or even multiple files), 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. In the following, if k is the identifier of a bug in DEFECTS4J, β_k denotes the buggy version of the code corresponding to bug k, ϕ_k denotes the code of the programmer-fixed version of β_k, and T_k denotes the tests accompanying k.

The bugs included in DEFECTS4J are a representative sample of real-world bugs, and as such they include several that admit simple fixes, as well as several others that require sweeping changes to different parts of a project. In order to focus the experiments on the bugs that have a chance of being in JAID's purview, we selected a subset of all bugs k that satisfy the following criteria: 1) the programmer-written fix ϕ_k only modifies Java executable code—no other artifacts like configuration files or compilation scripts; 2) the programmer-written fix ϕ_k modifies no more than 5 consecutive lines of code and no more than 4 statements with respect to β_k (as reported by the ChangeDistiller tool [8]); 3) the bug is reproducible: at least one test in T_k fails on β_k, and all tests in T_k pass on ϕ_k. A total of 138 bugs satisfy these criteria; these are the subjects of our experiments with JAID. Tab. I shows the size of and the number of bugs in each DEFECTS4J project among our experimental subjects.

B. Research Questions

Our evaluation addresses research questions in different areas:

Effectiveness: How many bugs can JAID fix?
Performance: How much time does JAID take?
Design: Which components of JAID's are the most important for effectiveness?
Comparison: How does JAID compare to other APR techniques for Java?

C. Setup

Since JAID ranks all generated snapshots according to their suspiciousness (see Sec. III-B), and depends on the ranking to guide the following stages, setting an arbitrary cutoff time may prevent from generating a complete ranking. Instead, we limit the search space in our experiments by configuring JAID so that it uses at most 1500 snapshots in order of suspiciousness;[3] then, the following stages (Fig. 1) all run to completion.

Each experiment targets one bug k in DEFECTS4J, and runs JAID on buggy code β_k using the tests T_k; the output is a ranked list of valid fixes for the bug. We manually inspect the top 50 fixes in order of ranking to determine which are *correct*; if all 50 fixes are incorrect, we continue the manual inspection of the other fixes and stop when we find a correct one, or no more valid fixes are available. We classify a fix as correct only if it is *semantically equivalent* to the programmer-written fix ϕ_k in DEFECTS4J. This is a high bar for correctness, which provides strong confidence that a fix is high-quality enough to be deployable.

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

[3]The number 1500 was chosen heuristically.

JDK 1.8. In this section, averages are measured using the median by default, with exceptions explicitly pointed out.

Other tools for Java APR. We quantitatively compare JAID to all other available tools for APR of Java programs that have also used DEFECTS4J in their evaluations:[4] 1) jGenProg is the implementation of GenProg [14], [33]—which works on C—for Java programs; we refer to jGenProg's evaluation in [19]; 2) jKali is the implementation of Kali [28]—which works on C—for Java programs; we refer to jKali's evaluation in [19]; 3) Nopol focuses on fixing Java conditional expression; we refer to Nopol's evaluation in [19]; 4) xPAR is a reimplementation of PAR [12]—which is not publicly available—discussed in [13] and [35]; 5) HDA implements the "history-driven" technique of [13]; 6) ACS implements the "precise condition synthesis" of [35].

Experiments with APR tools often target a different subset of DEFECTS4J that is amenable to the technique being evaluated. Comparing the number and kinds of fixed bugs among tools remains meaningful, because bugs that are excluded a priori from an evaluation can normally be considered beyond a tool's current capabilities.

D. Results

Effectiveness. JAID was able to produce valid fixes for the 31 bugs of DEFECTS4J listed in Tab. II. More significant, it produced *correct* fixes—equivalent to those written by programmers—for 25 of these bugs (integer RANK in Tab. II). This indicates that JAID is applicable to realistic code and, when it runs successfully, it often produces fixes of high quality. As we discuss in more detail below, the number of correctly fixed bugs is on par with, or above, the state of the art of Java APR techniques.

Unsurprisingly, JAID produces fixes that tend to be small in size: 1.7 lines of code changes per valid fix on average. This is a result of its fix generation process, which is based on state-based information, targets simple fix actions, and then sharpens their precision by injecting them into the code using conditional schemas.

When it is successful, JAID often produces a significant number of valid fixes—39 per fixed bug on average in our experiments—and a much smaller number of correct fixes—1 per fixed bug on average in our experiments—which all are semantically equivalent. In JAID's output for the 25 bugs that were correctly fixed, the median position of the first correct fix in the list of all valid fixes was 6 (from the top); for 15 bugs that JAID correctly fixed, a correct fix appears among the top-10 valid ones in order of ranking; for 9 bugs it appears at the top position. These results indicate the importance of ranking to ensure that the correct fixes are easier to spot among several valid but incorrect ones. For 10 bugs, the correct fix appears further down in the output list; in 6 of these cases, the correct fix turns out to be a "syntactic" one, but several valid, incorrect "semantic" fixes are generated and ranked higher. This suggests that improving the precision of ranking may benefit from mining additional information about common features of programmer-written fixes, as done by HDA and ACS.

JAID can correctly fix even 4 bugs that include only one failing test (and no passing tests), ranking the correct fix first in two cases. These results showcase how JAID can be successful at mitigating the baneful problem of overfitting.

Performance. As shown in Tab. II, JAID runs in 119.5 minutes per bug on average (median, whereas the mean running time is 355.1 minutes). JAID is unsurprisingly significantly slower than tools based on constraint solving and other symbolic techniques; for example, Nopol takes around 22 minutes per bug on average—on what, we assume, is comparable hardware. They are, however, in line with other APR techniques mainly based on dynamic analysis, such as jGenProg which takes about one hour per bug.

Looking more closely into how much time each stage of JAID takes, it is clear that *validation* is by far the most time consuming: fault localization takes 2.7% of the median time per bug, fix generation takes 0.5%, and fix validation takes 92.8%. Validation time tends to be proportional to the number of available tests, and to the number of fix candidates—which, in turn, is proportional to the number of snapshots that are actually analyzed. The approach outlined in Sec. III-E still helps save a significant amount of compilation time; as future work, we plan to further improve the performance of validation by running multiple concurrent instances on the same JVM.

Design. Which of the fix actions and schemas are the most useful to build correct fixes? Regarding fix actions, we distinguish three kinds: 1) s for actions modifying the state or an expression—"semantic" actions that are built on JAID's rich state-based abstractions; 2) m for actions mutating a statement—"syntactic" actions that correct common errors and are commonly used in APR systems; 3) c for actions modifying the control-flow—"terminating" actions that can still have a significant impact on program behavior. JAID uses all three kinds of actions, with similar frequencies, in correct fixes, which indicates that they are largely complementary and all contribute to JAID's effectiveness.

The five fix schemas of Fig. 2, which JAID uses to inject a fix action into the method under repair, also all feature in correct fixes. This suggests that both conditional and unconditional applications are required to target a wide choice of bugs.

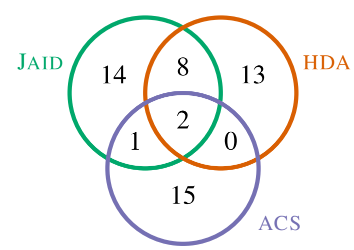

Fig. 3. Number of bugs correctly fixed by each of the main APR tool.

Comparison: correct fixes. Tab. III compares JAID to

[4]We ascertained that the version of DEFECTS4J used in our experiments does not differ substantially from those used in the other tools' experiments; in particular, all bugs analyzed by JAID were also available to the other tools.

TABLE II

Summary of the experimental results. For every bug (BUG ID k) that JAID fixed with a valid fix: the lines of code (LOC) of the method under repair; how many tests (TESTS), Passing and Failing, exercise the method under repair; the number of suspicious SNAPshots that are analyzed (capped at 1500); the number of CANDidate fixes that undergo validation; the mean SIZE, in lines of code changes, of a valid fix; the total number of VALID fixes; the number of CORRECT fixes among the top 50 valid fixes; the position of the first correct fix in ranking order (RANK); the TOTAL running time in minutes, and the breakdown into time for fault Localization, fix Generation, and fix Validation; the kinds of ACTIONS (s for state- and expression-modifying actions, m for mutation actions, and c for control-flow actions, see Sec. III-C), and the kinds of SCHEMAS (see Fig. 2) used in correct fixes.

			TESTS		GENERATION		FIXES				TIME					
BUG ID	k	LOC	P	F	SNAP	CAND	SIZE	VALID	CORRECT	RANK	TOTAL	L	G	V	ACTIONS	SCHEMAS
Chart	1	32	37	1	791	3762	1.5	536	0	84	54.1	1.0	1.5	51.6	m	E
Chart	24	6	0	1	813	2476	1.5	2	2	1	16.8	0.3	0.2	16.3	s	A,B
Chart	26	108	23	22	1500	2018	1.7	82	3	1	53.6	10.3	4.7	38.6	c, s	B,E
Chart	9	38	1	1	1500	5991	1.8	52	2	43	72.2	2.6	0.8	68.8	s	E
Closure	125	15	538	1	154	517	1.9	98	0	–	131.3	9.7	0.0	121.6	–	–
Closure	126	95	71	2	1500	4583	1.9	425	0	93	601.4	6.0	0.5	594.9	m	E
Closure	18	122	2096	1	1000	9215	1.0	9	1	1	1367.1	449.5	5.8	911.8	m	E
Closure	31	122	2037	1	1500	14464	1.0	9	1	8	1440.1	448.4	6.9	984.7	m	E
Closure	33	27	259	1	1500	4484	1.5	2720	2	1	258.0	6.5	0.1	251.4	c	B
Closure	40	46	305	2	871	5243	1.0	4	4	1	119.5	8.1	0.6	110.9	m	E
Closure	5	98	56	1	1500	25816	1.5	2	0	–	975.9	6.0	3.7	966.2	–	–
Closure	62	45	45	2	1500	7138	1.0	87	1	31	126.7	6.8	1.1	118.8	m	E
Closure	63	45	45	2	1500	7138	1.0	87	1	31	127.1	6.7	1.1	119.3	m	E
Closure	70	19	2337	5	393	2359	1.0	5	5	1	70.4	31.1	0.2	39.1	m	E
Closure	73	70	482	1	1500	11472	1.0	1	1	1	473.4	165.1	2.5	305.8	m	E
Lang	24	102	0	1	1500	51872	1.0	2	0	–	2228.6	1.2	7.2	2220.1	–	–
Lang	33	11	0	1	150	792	2.1	7	7	1	11.0	3.3	0.1	7.7	c, s	C,D,B
Lang	38	6	33	1	328	1363	1.7	28	4	4	10.7	1.0	0.1	9.6	s	A,B
Lang	39	126	1	1	1500	11702	2.1	39	0	–	408.2	10.7	3.7	393.9	–	–
Lang	45	37	0	1	1500	7173	1.9	68	2	34	105.1	0.7	0.6	103.9	s	B
Lang	51	51	0	1	959	8514	1.7	424	1	46	188.4	0.3	0.8	187.3	c	B
Lang	55	6	4	1	21	170	2.0	15	1	6	3.6	0.4	0.0	3.2	s	C
Lang	61	27	7	2	1500	18289	1.0	4	0	–	327.0	0.5	1.7	324.7	–	–
Math	105	2	8	1	64	1478	2.0	139	0	–	9.2	0.2	0.1	9.0	–	–
Math	32	52	6	1	1500	2997	1.0	5	1	4	37.5	1.7	0.4	35.4	s	E
Math	5	22	5	1	162	1426	2.0	61	3	1	11.3	0.5	0.1	10.8	c	B
Math	50	125	3	1	1500	37848	1.5	1101	3	28	1502.6	23.4	7.6	1471.5	s, c, m	E,C,D
Math	53	5	19	1	246	2010	2.0	10	2	6	19.0	3.2	0.1	15.7	c	B
Math	80	15	16	1	1500	9526	1.9	3877	0	1366	156.7	0.5	0.9	155.2	s	B,A
Math	82	15	13	1	436	1707	1.9	13	1	9	33.1	3.0	0.1	30.0	m	E
Math	85	43	12	1	1500	2922	1.7	709	0	247	68.3	1.2	0.2	66.9	m	E
MEDIAN		38	16	1	1500	4583	1.7	39	1	6	119.5	3.2	0.6	110.9		

TABLE III

A comparison of APR techniques on bugs in DEFECTS4J. This paper's JAID is compared to ACS, HDA, xPAR, jGenProg, and jKali. For each tool, the table reports the number of bugs that were fixed with a VALID fix; the number of bugs that were fixed with a CORRECT fix (among ANY of those outputed by the tool, only among the TOP-10 POSITIONS in the output, and only in the FIRST POSITION in the output); and the resulting PRECISION (CORRECT/VALID) and RECALL (CORRECT/357, where 357 is the total number of bugs in DEFECTS4J). Question marks represent data not available for a tool.

		ANY POSITION			TOP-10 POSITIONS			FIRST POSITION		
TOOL	VALID	CORRECT	PRECISION	RECALL	CORRECT	PRECISION	RECALL	CORRECT	PRECISION	RECALL
JAID	31	25	80.6	7.0	15	48.4	4.2	9	29.0	2.5
ACS	23	18	78.3	5.0	18	78.3	5.0	18	78.3	5.0
HDA	?	23	?	6.4	23	?	6.4	13	?	3.6
Nopol	35	5	14.3	1.4	5	14.3	1.4	5	14.3	1.4
xPAR	?	4	?	1.1	4	?	1.1	?	?	?
jGenProg	27	5	18.5	1.4	5	18.5	1.4	5	18.5	1.4
jKali	22	1	4.6	0.3	1	4.6	0.3	1	4.6	0.3

six other APR tools for Java. In terms of number of bugs fixed with a correct fix, JAID outperforms all other tools. Note that both runners-up, HDA and ACS, crucially rely on *mining additional information* from other sources: HDA mines frequency information about 3000 bug fixes from 800 popular GitHub projects, whereas ACS searches for predicates in "all open-source projects in GitHub" [35, Sec.III-E]. The implementation of HDA additionally requires fault localization information as part of the input. Thus, both tools use a richer input than just a buggy program and its accompanying unit tests, which indicates that JAID's performance is highly competitive, and arguably improving the state-of-the-art in its own league.

JAID fares very well also in terms of *precision* (fraction of bugs with a valid fix that have a correct fix) and *recall* (percentage of all bugs in DEFECTS4J that have a correct fix). Since different approaches, and different experimental evaluations, deal differently with bugs that admit multiple valid fixes, we measure three variants of precision and recall: 1) relative to the number of bugs that were correctly fixed by any of the valid fixes, regardless of the correct fix's rank; 2) relative to the number of bugs that were correctly fixed by a fix ranked among the top 10 in a tool's output; 3) relative to the number of bugs that were correctly fixed by a fix ranked first in a tool's output. JAID achieve the best precision[5] and recall if we disregard ranking; and the second-best precision and third-best recall in the other two cases. Again, note that the only tools that outperform JAID rely on additional input information to sharpen their precision and recall.

Comparison: kinds of fixes. Fig. 3 zooms in on the bugs that are correctly fixed by JAID, HDA, and ACS, and shows how many bugs each tool can fix that the others cannot. The tools are mainly *complementary* in the specific bugs they are successful on: JAID fixes 14 bugs that no other tool can fix; HDA fixes 13; and ACS fixes 15.

Among the tools not in Fig. 3: Nopol fixes 2 bugs that no other tool can fix (plus 2 bug also fixed by JAID, and 1 of which also fixed by HDA); jGenProg fixes 1 bug that no other tool can fix (plus 4 bugs also fixed by HDA, 3 of which JAID can also fix); jKali fixes 1 bug that jGenProg, Nopol, HDA and JAID can also fix. These numbers indicate that each technique is successful in its own domain. The complementarity also suggests that *combining* techniques based on mining (such as HDA and ACS) with JAID's techniques is likely to yield further improvements in terms of precision and effectiveness.

Comparison: other tools. We refrain from quantitatively comparing APR tools that target other programming languages—and thus were evaluated on different benchmarks [15]. Nevertheless, just to give an idea, Angelix [22] and Prophet [17][6] 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 [14]; AutoFix [25][7] achieves a precision of 59.3% and a recall of 25% on 204 bugs from various Eiffel projects with contracts.

E. 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 for the same bug, we classify it as incorrect. This approach is consistent with what done by other researchers. A more detailed analysis of patch correctness belongs to future work.

We measured, and compared, precision and recall relative to *all bugs* in DEFECTS4J, even if most APR techniques—including JAID—only run experiments on a subset of the 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 done in other evaluations [14], [17], [25], [28], [29].

Internal validity indicates whether the experimental results soundly support the findings. Comparing the 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 to quantitative results that are representative of a typical run [1]. Some techniques, such as ACS and HDA, 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 input to HDA's main algorithm. 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. 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. These characteristics 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 different projects as well.

V. RELATED WORK

Automated program repair has become a bustling research area in the course of just a few years. The first APR techniques [2], [33] used genetic algorithms to search the space of possible fixes for a valid one. GenProg [33] pioneered the "generate-and-validate" approach, where many plausible

[5]The precision of HDA is not reported in [13].
[6]Valid fixes are called "plausible" in [17].
[7]Correct fixes are called "proper" in [25], [26].

fixes are generated based on heuristics, and then are validated against the available tests. More recently, others [5], [21], [22], [24], [36] 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. For a broader list of APR techniques, see Monperrus's annotated bibliography [23].

Generate-and-validate. GenProg [33] 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 [20] that existing code is also applicable to patch incorrect functionality. GenProg's algorithm and implementation were substantially extended [14] to scale to code bases of realistic code sizes—producing valid fixes for 52% of 105 bugs.

Encouraged by GenProg's promising results, various approaches tried to make the mutation of candidate fixes more effective, or the search in the space of possible fixes more directed and thus more efficient. For example, MutRepair [4] only modifies operators appearing in expressions (such as comparison operators and Boolean connectives), since these tend to be a common source of programming mistakes. PAR [12] 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. A complementary approach [30] suggests to use *anti*-patterns, trying to capture fixes that are likely to be incorrect but still pass validation.

The overfitting problem. A more detailed analysis [28] of the fixes produced by GenProg and similar techniques has shown that only a small fraction of them is genuinely correct; for example, less than 2% of the bugs of [14] are correctly fixed. [28]'s analysis has pushed the research in APR to addressing this manifestation of the overfitting problem [29].

Most techniques for APR are based on tests, which are necessarily incomplete characterization of correct behavior. By also relying on contracts (specifications embedded in the program text) AutoFix [25], [26], [31] was the first general-purpose APR technique to substantially increase the number of correct fixes—for 25% of 204 bugs in [25]. JAID generalizes 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. SearchRepair [11] is one of few other approaches based on semantic analysis—as opposed to the more commonly used syntactic analysis. SearchRepair relies on preprocessing a large dataset of programmer-written code snippets, and encoding their behavior as input/output relational constraints; it then generates fixes by searching the dataset for snippets that capture the desired input/output behavior. HDA [13] 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. 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 often target the synthesis of *conditions* in if statements or loops, since changing those conditions often affects the control flow in decisive ways. SemFix [24] is one of the early examples; it relies on symbolic execution to summarize tests and on location-based fault localization, and it synthesizes expressions in conditionals and in assignments that try to avoid triggering failures. DirectFix [21] expresses the repair problem as a MaxSMT constraint, and supports generating multi-line fixes. Both SemFix and DirectFix, however, have limited scalability. Angelix [22] addresses this problem by introducing an efficient representation of constraints, and by combining it with a symbolic execution analysis similar to SemFix's.

Nopol [36] only targets conditional expressions, and uses a form of angelic debugging [3], [37] 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 [16] also combines 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. Prophet [17] improves SPR with a probabilistic model learned by mining programmer-written fixes. MintHint [10] also builds a statistical model to generate repair *suggestions* consisting of expressions that may be useful in a complete fix. ACS [35] is a recent technique that 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. JAID also relies on data- and control-dependency analysis, and can guess modifications to conditional expressions, but it does not need any additional source of information other than the project being fixed.

Runtime patching. Runtime patching [6], [7], [18] denotes approaches that operate at runtime as fallback measures in response to triggered failures—in contrast to APR techniques that modify source code. Under the hood, runtime patching often uses program analysis techniques similar to those of APR systems; ClearView [27], for instance, dynamically infer state invariants like JAID does, but does so at runtime on instrumented binaries with the goal of preventing problems such as buffer overflows.

ACKNOWLEDGMENTS

This work was partially supported by Hong Kong RGC General Research Fund (GRF) PolyU 152703/16E and the Hong Kong Polytechnic University internal fund 1-ZVJ1.

REFERENCES

[1] A. Arcuri and L. C. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Softw. Test., Verif. Reliab.*, 24(3):219–250, 2014.

[2] 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)*, pages 162–168. IEEE, 2008.

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

[4] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *3rd International Conference on Software Testing, Verification and Validation (ICST)*, pages 65–74. IEEE, 2010.

[5] 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*, pages 30–39. ACM, 2014.

[6] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. *ACM SIGPLAN Notices*, 38(11):78–95, 2003.

[7] B. Elkarablieh and S. Khurshid. Juzi: a tool for repairing complex data structures. In *Proceedings of the 30th International Conference on Software Engineering*, pages 855–858. ACM, 2008.

[8] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, Nov. 2007.

[9] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014. http://defects4j.org.

[10] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso. MintHint: automated synthesis of repair hints. In *36th International Conference on Software Engineering (ICSE)*, pages 266–276. ACM, 2014.

[11] 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)*, pages 295–306. IEEE, 2015.

[12] 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)*, pages 802–811. IEEE, 2013.

[13] 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)*, volume 1, pages 213–224. IEEE, 2016.

[14] 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)*, pages 3–13. IEEE, 2012.

[15] 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.*, 41(12):1236–1256, 2015.

[16] 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)*, pages 166–178. ACM, 2015.

[17] 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)*, pages 298–312. ACM, 2016.

[18] M. Z. Malik and K. Ghori. A case for automated debugging using data structure repair. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 620–624. IEEE, 2009.

[19] 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.

[20] 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)*, pages 492–495. ACM, 2014.

[21] S. Mechtaev, J. Yi, and A. Roychoudhury. DirectFix: Looking for simple program repairs. In *37th International Conference on Software Engineering (ICSE)*, volume 1, pages 448–458. IEEE, 2015.

[22] 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*, pages 691–701. ACM, 2016.

[23] M. Monperrus. Automatic software repair: a bibliography. Technical Report hal-01206501, University of Lille, 2015.

[24] 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)*, pages 772–781. IEEE, 2013.

[25] 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*, 40(5):427–449, 2014.

[26] 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)*, pages 392–395. IEEE, 2011.

[27] J. H. Perkins, G. Sullivan, W. Wong, Y. Zibin, M. D. Ernst, M. Rinard, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, and S. Sidiroglou. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*, pages 87–102, 2009.

[28] 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)*, pages 24–36. ACM, 2015.

[29] 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)*, pages 532–543. ACM, 2015.

[30] 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)*, pages 727–738. ACM, 2016.

[31] 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)*, pages 61–72. ACM, 2010.

[32] 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)*, pages 356–366. IEEE, 2013.

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

[34] W. E. Wong, V. Debroy, and B. Choi. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software*, 83(2):188–208, 2010.

[35] 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.

[36] 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*, 43(1):34–55, 2017.

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