Automated Fixing of Programs with Contracts

Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller, *Member, IEEE*

Abstract—This paper describes AutoFix, an automatic debugging technique that can fix faults in general-purpose software. To provide high-quality fix suggestions and to enable automation of the whole debugging process, AutoFix relies on the presence of simple specification elements in the form of contracts (such as pre- and postconditions). Using contracts enhances the precision of dynamic analysis techniques for fault detection and localization, and for validating fixes. The only required user input to the AutoFix supporting tool is then a faulty program annotated with contracts; the tool produces a collection of validated fixes for the fault ranked according to an estimate of their suitability. In an extensive experimental evaluation, we applied AutoFix to over 200 faults in four code bases of different maturity and quality (of implementation and of contracts). AutoFix successfully fixed 42 percent of the faults, producing, in the majority of cases, corrections of quality comparable to those competent programmers would write; the used computational resources were modest, with an average time per fix below 20 minutes on commodity hardware. These figures compare favorably to the state of the art in automated program fixing, and demonstrate that the AutoFix approach is successfully applicable to reduce the debugging burden in real-world scenarios.

Index Terms—Automatic program repair, contracts, dynamic analysis

1 Introduction

The programmer's ever recommencing fight against errors involves two tasks: finding faults; and correcting them. Both are in dire need of at least partial automation.

Techniques to *detect* errors automatically are becoming increasingly available and slowly making their way into industrial practice [1], [2], [3]. In contrast, automating the whole debugging process—in particular, the synthesis of suitable fixes—is still a challenging problem, and only recently have usable techniques (reviewed in Section 6) started to appear.

AutoFix, described in this paper, is a technique and supporting tool that can generate corrections for faults of general-purpose software[1] completely automatically. AutoFix targets programs annotated with *contracts*—simple specification elements in the form of preconditions, postconditions, and class invariants. Contracts provide a specification of correct behavior that can be used not only to detect faults automatically [4] but also to suggest corrections. The current implementation of AutoFix is integrated in the open-source eiffel verification environment [5]—the research branch of the EiffelStudio IDE—and works on programs written in Eiffel; its concepts and techniques are, however, applicable to any programming language supporting some form of annotations (such as JML for Java or the .NET CodeContracts libraries).

AutoFix combines various program analysis techniques—such as dynamic invariant inference, simple static analysis, and fault localization—and produces a collection of suggested fixes, ranked according to a heuristic measurement of relevance. The dynamic analysis for each fault is driven by a set of test cases that exercise the routine (method) where the fault occurs. While the AutoFix techniques are independent of how these test cases have been obtained, all our experiments so far have relied on the AutoTest random-testing framework to generate the test cases, using the contracts as oracles. This makes for a completely automatic debugging process that goes from detecting a fault to suggesting a patch for it. The only user input is a program annotated with the same contracts that programmers using a contract-equipped language normally write [6], [7].

In previous work, we presented the basic algorithms behind AutoFix and demonstrated them on some preliminary examples [8], [9]. The present paper discusses the latest AutoFix implementation, which combines and integrates the previous approaches to improve the flexibility and generality of the overall fixing technique. The paper also includes, in Section 5, an extensive experimental evaluation that applied AutoFix to over 200 faults in four code bases, including both open-source software developed by professionals and student projects of various quality. AutoFix successfully fixed 86 (or 42 percent) of the faults; inspection shows that 51 of these fixes are genuine corrections of quality comparable to those competent programmers would write. The other 35 fixes are not as satisfactory—because they may change the intended program behavior—but are still useful patches that pass all available regression tests; hence, they avoid program failure and can be used as suggestions for further debugging. AutoFix required only limited computational resources to produce the fixes, with an average time per fix below 20 minutes on commodity hardware (about half of the

1. As opposed to the domain-specific programs targeted by related repair techniques, which we review in Section 6.2.

- *Y. Pei, C. A. Furia, M. Nordio, and B. Meyer are with the Chair of Software Engineering, Department of Computer Science, ETH Zürich, Switzerland.*
- *Y. Wei is with the Constraint Reasoning Group, Microsoft Research Cambridge, United Kingdom.*
- *A. Zeller is with the Software Engineering Chair, Saarland University, Germany.*

Manuscript received 31 May 2013; revised 6 Feb. 2014; accepted 3 Mar. 2014. Date of publication 19 Mar. 2014; date of current version 14 May 2014. Recommended for acceptance by F. Tip.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2014.2312918

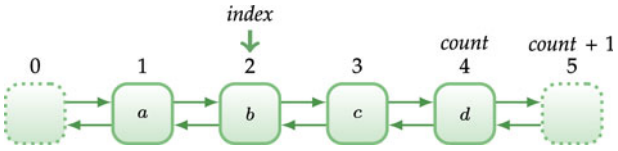


Fig. 1. A doubly-linked list implementing *TWO_WAY_SORTED_SET*. The cursor *index* is on position 2. The elements are stored in positions 1 to 4, whereas positions 0 *(before)* and 5 *(after)* mark the list's boundaries. *count* denotes the number of stored elements (i.e., four).

time is used to generate the test cases that expose the fault). These results provide strong evidence that AutoFix is a promising technique that can correct many faults found in real programs completely automatically, often with high reliability and modest computational resources.

In the rest of the paper, Section 2 gives an overview of AutoFix from a user's perspective, presenting a fault fixed automatically; the fault is included in the evaluation (Section 5) and is used as running example. Section 3 introduces some concepts and notation repeatedly used in the rest of the paper, such as the semantics of contracts and the program expressions manipulated by AutoFix. Section 4 presents the AutoFix algorithm in detail through its successive stages: program state abstraction, fault localization, synthesis of fix actions, generation of candidate fixes, validation of candidates, and ranking heuristics. Section 5 discusses the experimental evaluation, including a detailed statistical analysis of numerous important measures. Section 6 presents related work and compares it with our contribution. Finally, Section 7 includes a summary and concluding remarks.

2 AutoFix in Action

We begin with a concise demonstration of how AutoFix, as seen from a user's perspective, fixes faults completely automatically.

2.1 Moving Items in Sorted Sets

Class *TWO_WAY_SORTED_SET* is the standard Eiffel implementation of sets using a doubly-linked list. Fig. 2 outlines features (members) of the class, some annotated with their pre- (**require**) and postconditions (**ensure**).[2] As pictured in Fig. 1, the integer attribute *index* is an internal cursor useful to navigate the content of the set: the set elements occupy positions 1 to *count* (another integer attribute, storing the total number of elements in the set), whereas the indexes 0 and $count + 1$ correspond to the positions *before* the first element and *after* the last. *before* and *after* are also Boolean argumentless queries (member functions) that return **True** when the cursor is in the corresponding boundary positions.

Fig. 2 also shows the complete implementation of routine *move_item*, which moves an element v (passed as argument) from its current (unique) position in the set to the immediate left of the internal cursor *index*. For example, if the list contains $\langle a, b, c, d \rangle$ and *index* is 2 upon invocation (as in Fig. 1), *move_item* (*d*) changes the list to $\langle a, d, b, c \rangle$.

2. All annotations were provided by developers as part of the library implementation.

```
1  index: INTEGER -- Position of internal cursor.
2
3  count: INTEGER -- Number of elements in the set.
4
5  before: BOOLEAN -- Is index = 0 ?
6    do Result := (index = 0) end
7
8  after: BOOLEAN -- Is index = count + 1 ?
9
10 off: BOOLEAN -- Is cursor before or after ?
11
12 item: G          -- Item at current cursor position.
13   require not off
14
15 forth            -- Move cursor forward by one.
16   require not after
17   ensure index = old index + 1
18
19 has (v: G): BOOLEAN -- Does the set contain v ?
20   ensure Result implies count ≠ 0
21
22 go_i_th (i: INTEGER) -- Move cursor to position i.
23   require 0 ≤ i ≤ count + 1
24
25 put_left (v: G) -- Insert v to the left of cursor.
26   require not before
27
28 move_item (v: G) -- Move v to the left of cursor.
29   require
30     v ≠ Void
31     has (v)
32   local idx: INTEGER ; found: BOOLEAN
33   do
34     idx := index
35     from start until found or after loop
36       found := (v = item)
37       if not found then forth end
38     end
39     check found and not after end
40     remove
41     go_i_th (idx)
42     put_left (v)
43   end
```

Fig. 2. Some features of class *TWO_WAY_SORTED_SET*.

move_item's precondition requires that the actual argument v be a valid reference (not **Void**, that is not *null*) to an element already stored in the set (*has*(v)). After saving the cursor position as the local variable *idx*, the loop in lines 35-38 performs a linear search for the element v using the internal cursor: when the loop terminates, *index* denotes v's position in the set. The three routine calls on lines 40-42 complete the work: *remove* takes v out of the set; *go_i_th* restores *index* to its original value saved in *idx*; *put_left* puts v back in the set to the left of the position *index*.

2.2 An Error in *move_item*

Running AutoTest on class *TWO_WAY_SORTED_SET* for only a few minutes exposes, completely automatically, an error in the implementation of *move_item*.

```
44      if idx > index then
45          idx := idx − 1
46      end
```

Fig. 3. Correction of the error in *move_item* automatically generated by AutoFix.

The error is due to the property that calling *remove* decrements the *count* of elements in the set by one. AutoTest produces a test that calls *move_item* when *index* equals *count* + 1; after v is removed, this value is not a valid position because it exceeds the new value of *count* by two, while a valid cursor ranges between 0 and *count* + 1. The test violates *go_i_th*'s precondition (line 23), which enforces the consistency constraint on *index*, when *move_item* calls it on line 41.

This fault is quite subtle, and the failing test represents only a special case of a more general faulty behavior that occurs whenever v appears in the set in a position to the left of the initial value of *index*: even if *index* $\leq$ *count* initially, *put_left* will insert v in the wrong position as a result of *remove* decrementing *count*—which indirectly shifts the index of every element after *index* to the left by one. For example, if *index* is 3 initially, calling *move_item* (d) on $\langle a, d, b, c\rangle$ changes the set to $\langle a, b, d, c\rangle$, but the correct behavior is leaving it unchanged. Such additional inputs leading to erroneous behavior go undetected by AutoTest because the developers of *TWO_WAY_SORTED_SET* provided an incomplete postcondition; the class lacks a query to characterize the fault condition in general terms.[3]

2.3 Automatic Correction of the Error in *move_item*

AutoFix collects the test cases generated by AutoTest that exercise routine *move_item*. Based on them, and on other information gathered by dynamic and static analysis, it produces, after running only a few minutes on commodity hardware without any user input, up to 10 suggestions of fixes for the error discussed. The suggestions include only *valid* fixes: fixes that pass all available tests targeting *move_item*. Among them, we find the "*proper*" fix in Fig. 3, which completely corrects the error in a way that makes us confident enough to deploy it in the program. The correction consists of inserting the lines 44-46 in Fig. 3 before the call to *go_i_th* on line 41 in Fig. 2. The condition *idx* > *index* holds precisely when v was initially in a position to the left of *index*; in this case, we must decrement *idx* by one to accommodate the decreased value of *count* after the call to *remove*. This fix completely corrects the error beyond the specific case reported by AutoTest, even though *move_item* has no postcondition that formalizes its intended behavior.

3 Preliminaries: Contracts, Tests, and Predicates

To identify faults, distinguish between correct and faulty input, and abstract the state of objects at runtime, AutoFix relies on basic concepts which will now be summarized.

3.1 Contracts and Correctness

AutoFix works on Eiffel classes equipped with *contracts* [13]. Contracts define the specification of a class and consist of *assertions*: preconditions (**require**), postconditions (**ensure**), intermediate assertions (**check**), and class invariants (translated for simplicity of presentation into additional pre- and postconditions in the examples of this paper). Each assertion consists of one or more *clauses*, implicitly conjoined and usually displayed on different lines; for example, *move_ item's* precondition has two clauses: $v \neq$ **Void** on line 30 and *has*(v) on line 31.

Contracts provide a criterion to determine the correctness of a routine: every execution of a routine starting in a state satisfying the precondition (and the class invariant) must terminate in a state satisfying the postcondition (and the class invariant); every intermediate assertion must hold in any execution that reaches it; every call to another routine must occur in a state satisfying the callee's precondition. Whenever one of these conditions is violated, we have a *fault*,[4] uniquely identified by a location in the routine where the violation occurred and by the specific contract clause that is violated. For example, the fault discussed in Section 2 occurs on line 42 in routine *move_item* and violates the single precondition clause of *put_left*.

3.2 Tests and Correctness

In this work, a test case t is a sequence of object creations and routine invocations on the objects; if r is the last routine called in t, we say that t is a *test case for* r. A test case is *passing* if it terminates without violating any contract and *failing* otherwise.[5]

Every session of automated program fixing takes as input a set T of test cases, partitioned into sets P (passing) and F (failing). Each session targets a single specific fault—identified by some failing location f in some routine r and by a violated contract clause c. When we want to make the targeted fault explicit, we write T_r, P_r, and $F_r^{f,c}$. For example, $F^{42,\,\textbf{not}\ before}_{move\_item}$ denotes a set of test cases all violating *put_left*'s precondition at line 42 in *move_item*.

The fixing algorithm described in Section 4 is independent of whether the test cases T are generated automatically or written manually. The experiments discussed in Section 5 all use the random testing framework AutoTest [4] developed in our previous work. Relying on AutoTest makes the whole process, from fault detection to fixing, completely automatic; our experiments show that even short AutoTest sessions are sufficient to produce suitable test cases that AutoFix can use for generating good-quality fixes.

3.3 Expressions and Predicates

AutoFix understands the causes of faults and builds fixes by constructing and analyzing a number of *abstractions* of the program states. Such abstractions are based on Boolean *predicates* that AutoFix collects from three basic sources:

3. Recent work [10], [11], [12] has led to new versions of the libraries with strong (often complete) contracts, capturing all relevant postcondition properties.

4. Since contracts provide a specification of correct behavior, contract violations are actual faults and not mere *failures*.

5. Since execution cannot continue after a failure, a test case can only fail in the last call.

- argumentless Boolean queries;
- expressions appearing in the program text or in contracts;
- Boolean combinations of basic predicates (previous two items).

3.3.1 Argumentless Boolean Queries

Classes are usually equipped with a set of argumentless Boolean-valued functions (called *Boolean queries* from now on), defining key properties of the object state: a list is empty or not, the cursor is on boundary positions or *before* the first element (*off* and *before* in Fig. 2), a checking account is overdrawn or not. For a routine r, Q_r denotes the set of all calls to public Boolean queries on objects visible in r's body or contracts.

Boolean queries characterize fundamental object properties. Hence, they are good candidates to provide useful characterizations of object states: being argumentless, they describe the object state *absolutely*, as opposed to in relation with some given arguments; they usually do not have preconditions, and hence are always defined; they are widely used in object-oriented design, which suggests that they model important properties of classes. Some of our previous work [14], [15] showed the effectiveness of Boolean queries as a guide to partitioning the state space for testing and other applications.

3.3.2 Program Expressions

In addition to programmer-written Boolean queries, it is useful to build additional predicates by combining expressions extracted from the program text of failing routines and from failing contract clauses. For a routine r and a contract clause c, the set $E_{r,c}$ denotes all *expressions* (of any type) that appear in r's body or in c. We normally compute the set $E_{r,c}$ for a clause c that fails in some execution of r; for illustrative purposes, however, consider the simple case of the routine *before* and the contract clause *index* > 1 in Fig. 2: $E_{before,\ index>1}$ consists of the expressions **Result**, *index*, *index* $= 0$, *index* $> 1, 0, 1$.

Then, with the goal of collecting additional expressions that are applicable in the context of a routine r for describing program state, the set $\mathbb{E}_{r,c}$ extends $E_{r,c}$ by *unfolding* [6]: $\mathbb{E}_{r,c}$ includes all elements in $E_{r,c}$ and, for every $e \in E_{r,c}$ of reference type t and for every argumentless query q applicable to objects of type t, $\mathbb{E}_{r,c}$ also includes the expression $e.q$ (a call of q on target e). In the example, $\mathbb{E}_{before,\mathrm{index}>1} = E_{before,index>1}$ because all the expressions in $E_{before,index>1}$ are of primitive type (integer or Boolean), but this will no longer be the case for assertions involving references.

Finally, we combine the expressions in $\mathbb{E}_{r,c}$ to form Boolean *predicates*; the resulting set is denoted $B_{r,c}$. The set $B_{r,c}$ contains all predicates built according to the following rules:

Boolean expressions. b, for every Boolean $b \in \mathbb{E}_{r,c}$ of Boolean type (including, in particular, the Boolean queries Q_r defined in Section 3.3.1);

Voidness checks. e = **Void**, for every $e \in \mathbb{E}_{r,c}$ of reference type;

Integer comparisons. $e \sim e'$, for every $e \in \mathbb{E}_{r,c}$ of integer type, every $e' \in \mathbb{E}_{r,c} \setminus \{e\} \cup \{0\}$ also of integer type,[6] and every comparison operator $\sim$ in $\{=, <, \leq\}$;

Complements. **not** p, for every $p \in B_{r,c}$.

In the example, $B_{before,index>1}$ contains **Result** and **not Result**, since **Result** has Boolean type; the comparisons $index < 0$, $index \leq 0$, $index = 0$, $index \neq 0$, $index \geq 0$, *and* $index > 0$; and the same comparisons between index and the constant 1.

3.3.3 Combinations of Basic Predicates

One final source of predicates comes from the observation that the values of Boolean expressions describing object states are often correlated. For example, *off* always returns **True** on an empty set (Fig. 2); thus, the implication *count* = 0 *implies off* describes a correlation between two predicates that partially characterizes the semantics of routine *off*.

Considering all possible implications between predicates is impractical and leads to a huge number of often irrelevant predicates. Instead, we define the set $\mathbb{P}_{r,c}$ as the superset of $B_{r,c}$ that also includes:

- All *implications* appearing in c, in *contracts* of r, or in contracts of any routine appearing in $B_{r,c}$;
- For every implication *a* **implies** *b* collected from contracts, its *mutations* **not** *a* **implies** *b*, *a* **implies not** *b*, *b* **implies** *a* obtained by negating the antecedent a, the consequent b, or both.

These implications are often helpful in capturing the object state in faulty runs.

The collection of implications and their mutations may contain *redundancies* in the form of implications that are co-implied (they are always both true or both false). Redundancies increase the size of the predicate set without providing additional information. To prune redundancies, we use the automated theorem prover Z3 [16]: we iteratively *remove* redundant implications until we reach a fixpoint. In the remainder, we assume $\mathbb{P}_{r,c}$ has pruned out redundant implications using this procedure.

4 How AutoFix Works

Fig. 4 summarizes the steps of AutoFix processing, from failure to fix. The following sections give the details.

AutoFix starts with a set of test cases, some passing and some failing, that expose a specific fault. The fault being fixed is characterized by a program location f and by a violated contract clause c (Section 3.2); the presentation in this section leaves f and c implicit whenever clear from the context. The notion of *snapshot* (described in Section 4.1) is the fundamental abstraction for characterizing and understanding the behavior of the program in the passing or failing test cases; AutoFix uses snapshots to model correct and incorrect behavior. Fixing a fault requires finding a suitable location where to modify the program to *remove* the source of the error. Since each snapshot refers to a specific program location, *fault localization* (described in Section 4.2) boils down to ranking

6. The constant 0 is always included because it is likely to expose relevant cases.

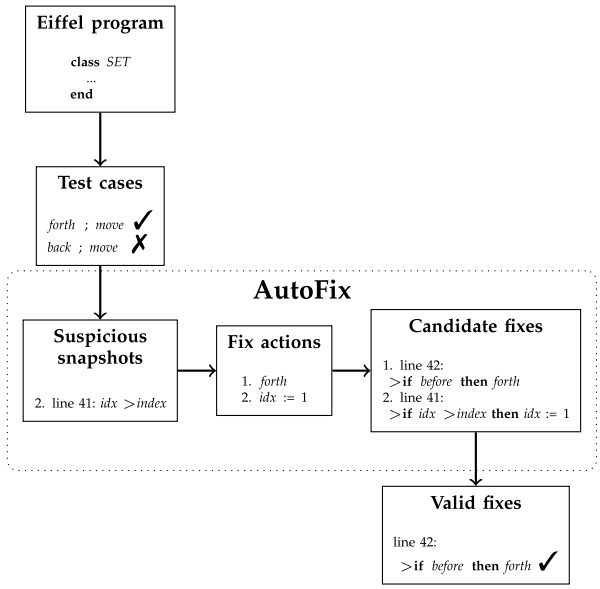


Fig. 4. How AutoFix works. Given an Eiffel program with contracts (Section 3.1), we generate passing and failing test cases that target a faulty routine (Section 3.2). By comparing the program state during passing and failing runs, AutoFix identifies *suspicious snapshots* (Sections 4.1–4.2) that denote likely locations and causes of failures. For each suspicious snapshot, AutoFix generates *fix actions* (Section 4.3) that can change the program state of the snapshot. Injecting fix actions into the original program determines a collection of *candidate fixes* (Section 4.4). The candidates that pass the regression test suite are *valid* (Section 4.5) and output to the user.

snapshots according to a combination of static and dynamic analyses that search for the origins of faults.

Once AutoFix has decided where to modify the program, it builds a code snippet that changes the program behavior at the chosen location. AutoFix synthesizes such *fix actions*, described in Section 4.3, by combining the information in snapshots with heuristics and behavioral abstractions that amend common sources of programming errors.

AutoFix injects fix actions at program locations according to simple conditional schema; the result is a collection of *candidate fixes* (Section 4.4). The following *validation* phase (Section 4.5) determines which candidate fixes pass all available test cases and can thus be retained.

In general, AutoFix builds several valid fixes for the same fault; the valid fixes are *ranked* according to heuristic measures of "quality" (Section 4.6), so that the best fixes are likely to emerge in top positions.

The latest implementation of AutoFix combines two approaches developed in previous work: model-based techniques [8] and code-based techniques [9].

4.1 Program State Abstraction: Snapshots

The first phase of the fixing algorithm constructs abstractions of the passing and failing runs that assess the program behavior in different conditions. These abstractions rely on the notion of *snapshot*[7]: a triple

$$\langle \ell, p, v \rangle,$$

consisting of a program location ℓ, a Boolean predicate p, and a Boolean value v. A snapshot abstracts one or more program executions that reach location ℓ with p evaluating to v. For example, $\langle 31, v = \mathbf{Void}, \mathbf{False} \rangle$ describes that the predicate $v = \mathbf{Void}$ evaluates to *False* in an execution reaching line 31.

Consider a routine r failing at some location f by violating a contract clause c. Given a set T_r of test cases for this fault, partitioned into passing P_r and failing $F_r^{f,c}$ as described in Section 3.2, AutoFix constructs a set $\mathsf{snap}(T_r)$ of snapshots. The snapshots come from two sources: invariant analysis (described in Section 4.1.1) and enumeration (Section 4.1.2).

We introduce some notation to define snapshots. A test case $t \in T_r$ describes a sequence $\mathsf{loc}(t) = \ell_1, \ell_2, \ldots$ of executed program locations. For an expression e and a location $\ell \in loc(t)$, $[\![e]\!]_t^\ell$ is the value of e at ℓ in t, if e can be evaluated at ℓ (otherwise, $[\![e]\!]_t^\ell$ is undefined).

4.1.1 Invariant Analysis

An *invariant* at a program location ℓ with respect to a set of test cases is a collection of predicates that all hold at ℓ in every run of the tests.[8] AutoFix uses Daikon [17] to infer invariants that characterize the *passing* and *failing* runs; their difference determine some snapshots that highlight possible failure causes.[9]

For each location ℓ reached by *some* tests in T_r, we compute the *passing invariant* π_ℓ as the collection of predicates that hold in all passing tests $P_r \subset T_r$; and the *failing invariant* ϕ_ℓ as the collection of predicates that hold in all failing tests in $F_r^{f,c} \subseteq T_r$. AutoFix uses only invariants built out of publicly visible predicates in $\mathbb{P}_{r,c}$. The predicates in $\Pi = \{p \,|\, p \in \phi_\ell \text{ and } \neg p \in \pi_\ell\}$ characterize potential causes of errors, as Π contains predicates that hold in failing runs but not in passing runs.[10] Correspondingly, the set $\mathsf{snap}(T_r)$ includes all components

$$\left\langle \ell, \bigwedge_{p \in \overline{\Pi}} p, \mathbf{True} \right\rangle,$$

for every non-empty subset $\overline{\Pi}$ of Π that profiles potential error causes.

The rationale for considering differences of sets of predicates is similar to the ideas behind the predicate elimination strategies in "cooperative bug isolation" techniques [18]. The dynamic analysis described in Section 4.2.2 would assign the highest dynamic score to snapshots whose predicates correspond to the deterministic bug predictors in cooperative bug isolation.

4.1.2 Enumeration

For each test $t \in T_r$, each predicate $p \in \mathbb{P}_{r,c}$, and each location $\ell \in \mathsf{loc}(t)$ reached in t's execution where the value of p

7. In previous work [9], we used the term "component" instead of "snapshot".

8. The class invariants mentioned in Section 3.1 are a special case.

9. Using Daikon is an implementation choice made to take advantage of its useful collection of invariant templates, which includes Boolean combinations beyond those described in Section 3.3.

10. Since the set of predicates used by AutoFix is closed under complement (Section 3.3), Π is equivalently computed as the negations of the predicates in $\{p \,|\, p \in \pi_\ell \text{ and } \neg p \in \phi_\ell\}$.

is defined, the set $\mathsf{snap}(T_r)$ of snapshots includes

$$\langle \ell, p, [\![p]\!]_t^\ell \rangle,$$

where p is evaluated at ℓ in t.

In the case of the fault of routine *move_item* (discussed in Section 2), the snapshots include, among many others, $\langle 34, v = \mathbf{Void}, \mathbf{False} \rangle$ (every execution has $v \neq \mathbf{Void}$ when it reaches line 34) and $\langle 41, idx > index, \mathbf{True} \rangle$ (executions failing at line 41 have *idx* > *index*).

Only considering snapshots corresponding to actual test executions avoids a blow-up in the size of $\mathsf{snap}(T_r)$. In our experiments (Section 5), the number of snapshots enumerated for each fault ranged from about a dozen to few hundreds; those achieving a high suspiciousness score (hence actually used to build fixes, as explained in Section 4.2.3) typically targeted only one or two locations ℓ with different predicates p.

4.2 Fault Localization

The goal of the fault localization phase is to determine which snapshots in $\mathsf{snap}(T_r)$ are reliable characterizations of the reasons for the fault under analysis. Fault localization in AutoFix computes a number of heuristic measures for each snapshot, described in the following sections; these include simple syntactic measures such as the distance between program statements (Section 4.2.1) and metrics based on the runtime behavior of the program in the passing and failing tests (Section 4.2.2).

The various measures are combined in a *ranking* of the snapshots (Section 4.2.3) to estimate their "*suspiciousness*": each triple $\langle \ell, p, v \rangle$ is assigned a score $\mathsf{susp}\langle \ell, p, v \rangle$ which assesses how suspicious the snapshot is. A high ranking for a snapshot $\langle \ell, p, v \rangle$ indicates that the fault is likely to originate at location ℓ when predicate p evaluates to v. The following phases of the fixing algorithm only target snapshots achieving a high score in the ranking.

4.2.1 Static Analysis

The static analysis performed by AutoFix is based on simple measures of proximity and similarity: *control dependence* measures the distance, in terms of number of instructions, between two program locations; *expression dependence* measures the syntactic similarity between two predicates. Both measures are variants of standard notions used in compiler construction [19], [20]. AutoFix uses control dependence to estimate the proximity of a location to where a contract violation is triggered; the algorithm then differentiates further among expressions evaluated at nearby program locations according to syntactic similarity between each expression and the violated contract clause. Static analysis provides coarse-grained measures that are only useful when combined with the more accurate dynamic analysis (Section 4.2.2) as described in Section 4.2.3.

Control dependence. AutoFix uses control dependence to rank locations (in snapshots) according to proximity to the location of failure. For two program locations ℓ_1, ℓ_2, write $\ell_1 \rightsquigarrow \ell_2$ if ℓ_1 and ℓ_2 belong to the same routine and there exists a directed path from ℓ_1 to ℓ_2 on the control-flow graph of the routine's body; otherwise, $\ell_1 \not\rightsquigarrow \ell_2$. The *control distance* $\mathsf{cdist}(\ell_1, \ell_2)$ of two program locations is the length of the shortest directed path from ℓ_1 to ℓ_2 on the control-flow graph if $\ell_1 \rightsquigarrow \ell_2$, and ∞ if $\ell_1 \not\rightsquigarrow \ell_2$. For example, $\mathsf{cdist}(40, 42) = 2$ in Fig. 2.

Correspondingly, when $\ell \rightsquigarrow \jmath$, the *control dependence* $\mathsf{cdep}(\ell, \jmath)$ is the normalized score:

$$\mathsf{cdep}(\ell, \jmath) = 1 - \frac{\mathsf{cdist}(\ell, \jmath)}{\max\{\mathsf{cdist}(\lambda, \jmath) \mid \lambda \in r \text{ and } \lambda \rightsquigarrow \jmath\}},$$

where λ ranges over all locations in routine r (where ℓ and $\jmath$ also appear); otherwise, $\ell \rightsquigarrow \jmath$ and $\mathsf{cdep}(\ell, \jmath) = 0$.

Ignoring whether a path in the control-flow graph is feasible when computing control-dependence scores does not affect the overall precision of AutoFix's heuristics: Section 4.2.3 shows how static analysis scores are combined with a score obtained by dynamic analysis; when the latter is zero (the case for unfeasible paths, which no test can exercise), the overall score is also zero regardless of static analysis scores.

Expression dependence. AutoFix uses expression dependence to rank expressions (in snapshots) according to similarity to the *contract clause* violated in a failure. Expression dependence is meaningful for expressions evaluated in the same local environment (that is, with strong control dependence), where the same syntax is likely to refer to identical program elements. Considering only syntactic similarity is sufficient because AutoFix will be able to affect the value of any assignable expressions (see Section 4.3). For an expression e, define the set $\mathsf{sub}(e)$ of its sub-expressions as follows:

- $e \in \mathsf{sub}(e)$;
- if $e' \in \mathsf{sub}(e)$ is a query call of the form $t.q\,(a_1, \ldots, a_m)$ for $m \geq 0$, then $t \in \mathsf{sub}(e)$ and $a_i \in \mathsf{sub}(e)$ for all $1 \leq i \leq m$.

This definition also accommodates infix operators (such as Boolean connectives and arithmetic operators), which are just syntactic sugar for query calls; for example a and b are both sub-expressions of $a + b$, a shorthand for *a.plus* (*b*). Unqualified query calls are treated as qualified call on the implicit target **Current**.

The *expression proximity* $\mathsf{eprox}(e_1, e_2)$ of two expressions e_1, e_2 measures how similar e_1 and e_2 are in terms of shared sub-expressions; namely, $\mathsf{eprox}(e_1, e_2) = |\mathsf{sub}(e_1) \cap \mathsf{sub}(e_2)|$. For example, the expression proximity $\mathsf{eprox}(i <= count,\ 0 \leq i \leq count + 1)$ is 2, corresponding to the shared sub-expressions *i* and *count*. The larger the expression proximity between two expressions is, the more similar they are.

Correspondingly, the *expression dependence* $\mathsf{edep}(p, c)$ is the normalized score:

$$\mathsf{edep}(p, c) = \frac{\mathsf{eprox}(p, c)}{\max\{\mathsf{eprox}(\pi, c) \mid \pi \in \mathbb{P}_{r,c}\}},$$

measuring the amount of evidence that p and c are syntactically similar. In routine *before* in Fig. 2, for example, $\mathsf{edep}(index, \mathrm{index} = 0)$ is $1/3$ because $\mathsf{eprox}(index, index = 0) = 1$ and $index = 0$ itself has the maximum expression proximity to $index = 0$.

4.2.2 Dynamic Analysis

Our dynamic analysis borrows techniques from generic fault localization [21] to determine which locations are likely to host the cause of failure. Each snapshot receives a *dynamic score* $\mathsf{dyn}\langle \ell, p, v\rangle$, roughly measuring how often it appears in failing runs as opposed to passing runs. A high dynamic score is empirical evidence that the snapshot characterizes the fault and suggests what has to be changed; we use static analysis (Section 4.2.1) to differentiate further among snapshots that receive similar dynamic scores.

Principles for computing the dynamic score. Consider a failure violating the contract clause c at location f in some routine r. For a test case $t \in T_r$ and a snapshot $\langle \ell, p, v\rangle$ such that ℓ is a location in r's body, write $\langle \ell, p, v\rangle \in t$ if t reaches location ℓ at least once and p evaluates to v there:

$$\langle \ell, p, v\rangle \in t \quad \text{iff} \quad \exists \ell_i \in \mathsf{loc}(t), \ell = \ell_i, \text{ and } v = [\![p]\!]_t^{\ell_i}.$$

For every test case $t \in T_r$ such that $\langle \ell, p, v\rangle \in t$, $\sigma(t)$ describes t's contribution to the dynamic score of $\langle \ell, p, v\rangle$: a large $\sigma(t)$ should denote evidence that $\langle \ell, p, v\rangle$ is a likely "source" of error if t is a failing test case, and evidence against it if t is passing. We choose a σ that meets the following requirements:

a) If there is at least one failing test case t such that $\langle \ell, p, v\rangle \in t$, the overall score assigned to $\langle \ell, p, v\rangle$ must be positive: the evidence provided by failing test cases cannot be canceled out completely.
b) The magnitude of each failing (resp. passing) test case's contribution $\sigma(t)$ to the dynamic score assigned to $\langle \ell, p, v\rangle$ decreases as more failing (resp. passing) test cases for that snapshot are available: the evidence provided by the first few test cases is crucial, while repeated outcomes carry a lower weight.
c) The evidence provided by one failing test case alone is stronger than the evidence provided by one passing test case.

The first two principles correspond to "Heuristic III" of Wong et al. [21], whose experiments yielded better fault localization accuracy than most alternative approaches. According to these principles, snapshots appearing only in failing test cases are more likely to be fault causes.

AutoFix's dynamic analysis assigns scores starting from the same basic principles as Wong et al.'s, but with differences suggested by the ultimate goal of automatic fixing: our dynamic score ranks snapshots rather than just program locations, and assigns weight to test cases differently. Contracts help find the location responsible for a fault: in many cases, it is close to where the contract violation occurred; on the other hand, automatic fixing requires gathering information not only about the location but also about the state "responsible" for the fault. This observation led to the application of fault localization principles on snapshots in AutoFix. It is also consistent with recent experimental evidence [22] suggesting that the behavior of existing fault localization techniques on the standard benchmarks used to evaluate them is not always a good predictor of their performance in the context of automated program repair; hence the necessity of adapting to the specific needs of automated fixing.[11]

Dynamic score. Assume an arbitrary order on the test cases and let $\sigma(t)$ be α^i for the ith failing test case t and $\beta\alpha^i$ for the ith passing test case. Selecting $0 < \alpha < 1$ decreases the contribution of each test case exponentially, which meets principle (b); then, selecting $0 < \beta < 1$ fulfills principle (c).

The evidence provided by each test case adds up:

$$\mathsf{dyn}\langle \ell, p, v\rangle = \gamma + \sum\{\sigma(u) \mid u \in F_r^{f,c}\} - \sum\{\sigma(v) \mid v \in P_r\},$$

for some $\gamma \geq 0$; the chosen ordering is immaterial. We compute the score with the closed form of geometric progressions:

$$\begin{aligned}
\#\mathsf{p}\langle \ell, p, v\rangle &= |\{t \in P_r \mid \langle \ell, p, v\rangle \in t\}|,\\
\#\mathsf{f}\langle \ell, p, v\rangle &= \left|\left\{t \in F_r^{f,c} \mid \langle \ell, p, v\rangle \in t\right\}\right|,\\
\mathsf{dyn}\langle \ell, p, v\rangle &= \gamma + \frac{\alpha}{1-\alpha}\left(1 - \beta + \beta\alpha^{\#\mathsf{p}\langle \ell,p,v\rangle} - \alpha^{\#\mathsf{f}\langle \ell,p,v\rangle}\right),
\end{aligned}$$

where $\#\mathsf{p}\langle \ell, p, v\rangle$ and $\#\mathsf{f}\langle \ell, p, v\rangle$ are the number of passing and failing test cases that determine the snapshot $\langle \ell, p, v\rangle$. It is straightforward to prove that $\mathsf{dyn}\langle \ell, p, v\rangle$ is positive if $\#\mathsf{f}\langle \ell, p, v\rangle \geq 1$, for every nonnegative α, β, γ such that $0 < \alpha + \beta < 1$; hence the score meets principle (a) as well.

Since the dynamic score dyn varies exponentially only with the number of passing and failing test cases, the overall success rate of the AutoFix algorithm is affected mainly by the number of tests but not significantly by variations in the values of α and β. A small empirical trial involving a sample of the faults used in the evaluation of Section 5 confirmed this expectation of robustness; it also suggested selecting the values $\alpha = 1/3$, $\beta = 2/3$, and $\gamma = 1$ as defaults in the current implementation of AutoFix, which tend to produce slightly shorter running times on average (up to 10 percent improvement). With these values, one can check that $2/3 < \mathsf{dyn}\langle \ell, p, v\rangle < 3/2$, and $1 < \mathsf{dyn}\langle \ell, p, v\rangle < 3/2$ if at least one failing test exercises the snapshot.

4.2.3 Overall Score

AutoFix combines the various metrics into an overall score $\mathsf{susp}\langle \ell, p, v\rangle$. The score puts together static and dynamic metrics with the idea that the latter give the primary source of evidence, whereas the less precise evidence provided by static analysis is useful to discriminate among snapshots with similar dynamic behavior.

Since the static measures are normalized ratios, and the dynamic score is also fractional, we may combine them by harmonic mean [23]:

$$\mathsf{susp}\langle \ell, p, v\rangle = \frac{3}{\mathsf{edep}(p, c)^{-1} + \mathsf{cdep}(\ell, f)^{-1} + \mathsf{dyn}\langle \ell, p, v\rangle^{-1}}.$$

Our current choice of parameters for the dynamic score (Section 4.2.2) makes it dominant in determining the overall

11. The results of Wong et al.'s heuristics in Qi et al.'s experiments [22] are not directly applicable to AutoFix (which uses different algorithms and adapts Wong et al.'s heuristics to its specific needs); replication belongs to future work.

score $\mathsf{susp}\langle \ell, p, v\rangle$: while expression and control dependence vary between 0 and 1, the dynamic score has minimum 1 (for at least one failing test case and indefinitely many passing). This range difference is consistent with the principle that dynamic analysis is the principal source of evidence.

For the fault of Fig. 2, the snapshot $\langle 41, idx > index, \mathbf{True}\rangle$ receives a high overall score. AutoFix targets snapshots such as this in the fix action phase.

4.3 Fix Action Synthesis

A snapshot $\langle \ell, p, v\rangle$ in $\mathsf{snap}(T_r)$ with a high score $\mathsf{susp}\langle \ell, p, v\rangle$ suggests that the "cause" of the fault under analysis is that expression p takes value v when the execution reaches ℓ. Correspondingly, AutoFix tries to build fixing *actions* (snippets of instructions) that *modify* the value of p at ℓ, so that the execution can hopefully continue without triggering the fault. This view reduces fixing to a program synthesis problem: find an action *snip* that satisfies the specification:

$$\textbf{require } p = v \textbf{ do } snip \textbf{ ensure } p \neq v \textbf{ end } .$$

AutoFix uses two basic strategies for generating fixing actions: setting and replacement. Setting (described in Section 4.3.1) consists of modifying the value of variables or objects through assignments or routine calls. Replacement (described in Section 4.3.2) consists of modifying the value of expressions directly where they are used in the program. Three simple heuristics, with increasing specificity, help prevent the combinatorial explosion in the generation of fixing actions:

1) Since the majority of program fixes are short and simple [24], [25], we only generate fixing actions that consist of simple instructions;
2) We select the instructions in the actions according to context (the location that we are fixing) and common patterns, and based on behavioral models of the classes (Section 4.3.3);
3) For integer expressions, we also deploy constraint solving techniques to build suitable derived expressions (Section 4.3.4).

We now describe actions by setting and replacements, which are the basic mechanisms AutoFix uses to synthesize actions, as well as the usage of behavioral models and constraint solving. To limit the number of candidates, AutoFix uses no more than one basic action in each candidate fix.

4.3.1 Actions by Setting

One way to change the value of a predicate is to modify the value of its constituent expressions by assigning new values to them or by calling modifier routines on them. For example, calling routine *forth* on the current object has the indirect effect of setting predicate *before* to **False**.

Not all expressions are directly modifiable by setting; an expression e is *modifiable* at a location ℓ if: e is of reference type (hence we can use e as target of routine calls); or e is of integer type and the assignment $e := 0$ can be executed at ℓ; or e is of Boolean type and the assignment e := **True** can be executed at ℓ. For example, *index* is modifiable everywhere in routine *move_item* because it is an attribute of the enclosing class; the argument *i* of routine *go_i_th*, instead, is not modifiable within its scope because arguments are read-only in Eiffel.

Since the Boolean predicates of snapshots may not be directly modifiable, we also consider sub-expressions of any type. The definition of sub-expression (introduced in Section 4.2.1) induces a partial order $\preceq$: $e_1 \preceq e_2$ iff $e_1 \in \mathsf{sub}(e_2)$ that is e_1 is a sub-expression of e_2; correspondingly, we define the *largest* expressions in a set as those that are only sub-expressions of themselves. For example, the largest expressions of integer type in $\mathsf{sub}(idx < index \ or \ after)$ are *idx* and *index*.

A snapshot $\langle \ell, p, v\rangle$ induces a set of target expressions that are modifiable in the context given by the snapshot. For each type (Boolean, integer, and reference), the set $\mathsf{targ}\langle \ell, p\rangle$ of *target expressions* includes the largest expressions of that type among p's sub-expressions $\mathsf{sub}(p)$ that are modifiable at ℓ. For example, $\mathsf{targ}\langle 41, idx > \mathbf{Current}.index\rangle$ in Fig. 2 includes the reference expression **Current**, the integer expressions **Current**.*index* and *idx*, but no Boolean expressions (*idx* > **Current**.*index* is not modifiable because it is not a valid L-value of an assignment).

Finally, the algorithm constructs the set $\mathsf{set}\langle \ell, p\rangle$ of *settings* induced by a snapshot $\langle \ell, p, v\rangle$ according to the target types as follows; these include elementary assignments, as well as the available routine calls.

Boolean targets. For $e \in \mathsf{targ}\langle \ell, p\rangle$ of Boolean type, $\mathsf{set}\langle \ell, p\rangle$ includes the assignments $e := d$ for d equal to the constants **True** and **False** and to the complement expression **not** *e*.

Integer targets. For $e \in \mathsf{targ}\langle \ell, p\rangle$ of integer type, $\mathsf{set}\langle \ell, p\rangle$ includes the assignments $e := d$ for d equal to the constants 0, 1, and -1, the "shifted" expressions $e + 1$ and $e - 1$, and the expressions deriving from integer constraint solving (discussed in Section 4.3.4).

Reference targets. For $e \in \mathsf{targ}\langle \ell, p\rangle$ of reference type, if $e.c\,(a_1, \ldots, a_n)$ is a call to a command (procedure) c executable at ℓ, include $e.c\,(a_1, \ldots, a_n)$ in $\mathsf{set}\langle \ell, p\rangle$. (Section 4.3.3 discusses how behavioral models help select executable calls at ℓ with chances of affecting the program state indicated by the snapshot.)

In the example of Section 2, the fault's snapshot $\langle 41, idx > index, \mathbf{True}\rangle$ determines the settings $\mathsf{set}\langle 41, idx > index\rangle$ that include assignments of 0, 1, and -1 to *idx* and *index*, and unit increments and decrements of the same variables.

4.3.2 Actions by Replacement

In some cases, assigning new values to an expression is undesirable or infeasible. For example, expression i in routine *go_i_th* of Fig. 2 does not have any modifiable sub-expression. In such situations, *replacement* directly substitutes the usage of expressions in existing instructions. Replacing the argument *idx* with *idx* - 1 on line 41 modifies the effect of the call to *go_i_th* without directly changing any local or global variables.

Every location ℓ labels either a primitive instruction (an assignment or a routine call) or a Boolean condition (the branching condition of an **if** instruction or the exit condition of a **loop**). Correspondingly, we define the set $\mathsf{sub}(\ell)$ of sub-expressions of a *location* ℓ as follows:

- if ℓ labels a Boolean condition b then $\mathsf{sub}(\ell) = \mathsf{sub}(b)$;
- if ℓ labels an assignment $v := e$ then $\mathsf{sub}(\ell) = \mathsf{sub}(e)$;
- if ℓ labels a routine call $t.c\,(a_1, \ldots, a_n)$ then

$$\mathsf{sub}(\ell) = \bigcup \{\mathsf{sub}(a_i) \mid 1 \leq i \leq n\}.$$

Then, a snapshot $\langle \ell, p, v \rangle$ determines a set $\mathsf{replace}\langle \ell, p \rangle$ of *replacements*: instructions obtained by replacing one of the sub-expressions of the instruction at ℓ according to the same simple heuristics used for setting. More precisely, we consider expressions e among the largest ones of Boolean or integer type in $\mathsf{sub}(p)$ and we modify their occurrences in the instruction at ℓ. Notice that if ℓ labels a conditional or loop, we replace e only in the Boolean condition, not in the body of the compound instruction.

Boolean expressions. For e of Boolean type, $\mathsf{replace}\langle \ell, p \rangle$ includes the instructions obtained by replacing each occurrence of e in ℓ by the constants **True** and **False** and by the complement expression **not** e.

Integer expressions. For e of integer type, $\mathsf{replace}\langle \ell, p \rangle$ includes the instructions obtained by replacing each occurrence of e in ℓ by the constants 0, 1, and -1, by the "shifted" expressions $e + 1$ and $e - 1$, and by the expressions deriving from integer constraint solving (Section 4.3.4).

Continuing the example of the fault of Section 2, the snapshot $\langle 41, idx > index, \mathbf{True} \rangle$ induces the replacement set $\mathsf{replace}\langle 41, idx > index \rangle$ including *go_i_th (idx−1)*, *go_i_th (idx + 1)*, as well as *go_i_th* (0), *go_i_th* (1), and *go_i_th* (-1).

4.3.3 Behavioral Models

Some of the fixing actions generated by AutoFix try to modify the program state by calling routines on the current or other objects. This generation is not blind but targets operations applicable to the target objects that can modify the value of the predicate p in the current snapshot $\langle \ell, p, v \rangle$. To this end, we exploit the *finite-state behavioral model* abstraction to quickly find out the most promising operations or operation sequences.

Using techniques we previously developed for Pachika [15], AutoFix extracts a simple behavioral model from *all* passing runs of the class under consideration. The behavioral model represents a *predicate abstraction* of the class behavior. It is a finite-state automaton whose states are labeled with predicates that hold in that state, and transitions are labeled with routine names, connecting observed pre-state to observed post-states.

As an example, Fig. 5 shows a partial behavioral model for the *forth* routine in Fig. 2. This behavioral model shows, among other things, that **not** *before* always holds after calls to *forth* in any valid initial state. By combining this information with the snapshot $\langle 42, before, \mathbf{True} \rangle$, we can surmise that invoking *forth* on line 42 mutates the current object state so that it avoids the possible failure cause *before* = **True**.

In general, the built behavioral abstraction is neither complete nor sound because it is based on a finite number of test runs. Nonetheless, it is often sufficiently precise to reduce the generation of routine calls to those that are likely to affect the snapshot state in the few cases where enumerating all actions by setting (Section 4.3.1) is impractical.

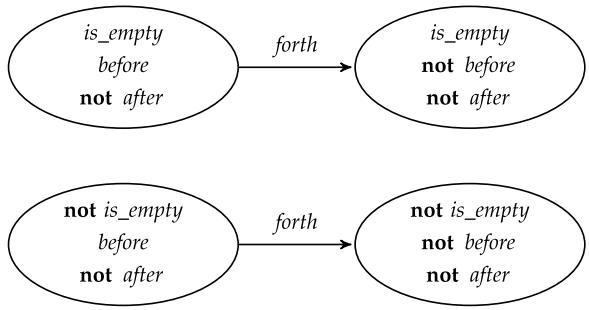


Fig. 5. Behavioral model of routine forth.

4.3.4 Constraint Solving

In contract-based development, numerous assertions take the form of Boolean combinations of linear inequalities over program variables and constants. The precondition of *go_i_th* on line 23 in Fig. 2 is an example of such *linearly constrained assertions* (or *linear assertions* for short). Such precondition requires that the argument *i* denote a valid position inside the set.

When dealing with integer expressions extracted from linear assertions, we deploy specific techniques to generate fixing actions in addition to the basic heuristics discussed in the previous sections (such as trying out the "special" values 0 and 1). The basic idea is to *solve* linear assertions for extremal values compatible with the constraint. Given a snapshot $\langle \ell, \lambda, v \rangle$ such that λ is a linear assertion, and an integer expression j appearing in λ, AutoFix uses Mathematica to solve λ for maximal and minimal values of j as a function of the other parameters (numeric or symbolic) in λ. To increase the quality of the solution, we strengthen λ with linear assertions from the class invariants that share identifiers with λ. In the example of *go_i_th*, the class invariant $count \geq 0$ would be added to λ when looking for extrema. The solution consists, in this case, of the extremal values 0 and $count + 1$, which are both used as replacements (Section 4.3.2) of variable *i*.

4.4 Candidate Fix Generation

Given a "suspicious" snapshot $\langle \ell, p, v \rangle$ in $\mathsf{snap}(T_r)$, the previous section showed how to generate fix actions that can mutate the value of p at location ℓ. Injecting any such fix actions at location ℓ gives a modified program that is a *candidate fix*: a program where the faulty behavior may have been corrected. We inject fix actions in program in two phases. First, we select a *fix schema*—a template that abstracts common instruction patterns (Section 4.4.1). Then, we *instantiate* the fix schema with the snapshot's predicate p and some fixing action it induces (Section 4.4.2).

Whereas the space of all possible fixes generated with this approach is potentially huge, AutoFix only generates candidate fixes for the few most suspicious snapshots (15 most suspicious ones, in the current implementation). In our experiments, each snapshot determines at most 50 candidate fixes (on average, no more than 30), which can be validated in reasonable time (see Section 5.3.3).

(*a*) *snippet*
old_stmt

(*b*) **if** *fail* **then**
snippet
end
old_stmt

(*c*) **if not** *fail* **then**
old_stmt
end

(*d*) **if** *fail* **then**
snippet
else
old_stmt
end

Fig. 6. Fix schemas implemented in AutoFix.

4.4.1 Fix Schemas

AutoFix uses a set of predefined templates called *fix schemas*. The four fix schemas currently supported are shown in Fig. 6;[12] they consist of conditional wrappers that apply the fix actions only in certain conditions (with the exception of schema *a* which is unconditional). In the schemas, *fail* is a placeholder for a predicate, *snippet* is a fixing action, and *old_stmt* are the statements in the original program where the fix is injected.

4.4.2 Schema Instantiation

For a state snapshot $\langle \ell, p, v \rangle$, we instantiate the schemas in Fig. 6 as follows:

fail becomes $p = v$, the snapshot's predicate and value.
snippet becomes any fix action by setting ($\mathsf{set}\langle \ell, p \rangle$ in Section 4.3.1) or by replacement ($\mathsf{replace}\langle \ell, p \rangle$ in Section 4.3.2).
old_stmt is the instruction at location ℓ in the original program.

The instantiated schema *replaces* the instruction at position ℓ in the program being fixed; the modified program is a *candidate fix*.

For example, consider again the snapshot $\langle 41, idx > index, \mathbf{True} \rangle$, which receives a high "suspiciousness" score for the fault described in Section 2 and which induces, among others, the fix action consisting of decrementing *idx*. The corresponding instantiation of fix schema (b) in Fig. 6 is then: *fail* becomes *idx* > *index* = **True**, *snippet* becomes *idx* := *idx* − 1, and *old_stmt* is the instruction *go_i_th (idx)* on line 23 in Fig. 2. Injecting the instantiated schema (replacing line 23) yields the candidate fix in Fig. 3, already discussed in Section 2.

4.5 Fix Validation

The generation of candidate fixes, described in the previous Sections 4.3 and 4.4, involves several heuristics and is "best effort": there is no guarantee that the candidates actually correct the error (or even that they are executable programs). Each candidate fix must pass a validation phase which determines whether its deployment removes the erroneous behavior under consideration. The validation phase regressively runs each candidate fix through the full set T_r of passing and failing test cases for the routine r being fixed. A fix is *validated* (or *valid*) if it passes all the previously failing test cases $F_r^{f,c}$ and it still passes the original passing test cases P_r. AutoFix only reports valid fixes to users, ranked as described in Section 4.6.

The correctness of a program is defined relative to its specification; in the case of automated program fixing, this implies that the validated fixes are only as good as the available tests or, if these are generated automatically, as the available contracts. In other words, evidently incomplete or incorrect contracts may let inappropriate candidate fixes pass the validation phase.

To distinguish between fixes that merely pass the validation phase because they do not violate any of the available contracts and high-quality fixes that developers would confidently deploy, we introduce the notion of *proper* fix. Intuitively, a proper fix is one that removes a fault without introducing other faulty or unexpected behavior. More rigorously, assume we have the complete behavioral specification $\mathfrak{S}_r$ of a routine r; following our related work [10], [12], $\mathfrak{S}_r$ is a pre-/postcondition pair that characterizes the effects of executing r on every query (attribute or function) of its enclosing class. A valid fix is *proper* if it satisfies $\mathfrak{S}_r$; conversely, it is *improper* if it is valid but not proper.

While we have demonstrated [12] that it is possible to formalize complete behavioral specifications in many interesting cases (in particular, for a large part of the EiffelBase library used in the experiments of Section 5), the line between proper and improper may be fuzzy under some circumstances when the notion of "reasonable" behavior is disputable or context-dependent. Conversely, there are cases—such as when building a proper fix is very complex or exceedingly expensive—where a valid but improper fix is still better than no fix at all because it removes a concrete failure and lets the program continue its execution.

In spite of these difficulties of principle, the experiments in Section 5 show that the simple contracts normally available in Eiffel programs are often good enough in many practical cases to enable AutoFix to suggest fixes that we can confidently classify as *proper*, as they meet the expectations of real programmers familiar with the code base under analysis.

4.6 Fix Ranking

The AutoFix algorithm often finds *several* valid fixes for a given fault. While it is ultimately the programmer's responsibility to select which one to deploy, flooding them with many fixes defeats the purpose of automated debugging, because understanding what the various fixes actually do and deciding which one is the most appropriate is tantamount to the effort of designing a fix in the first place.

To facilitate the selection, AutoFix ranks the valid fixes according to the "suspiciousness" score $\mathsf{susp}\langle \ell, p, v \rangle$ of the snapshot $\langle \ell, p, v \rangle$ that determined each fix.[13] Since multiple fixing actions may determine valid fixes for the same snapshot, ties in the ranking are possible. The experiments in Section 5 demonstrate that high-quality proper fixes often

12. Recent work [25] has demonstrated that these simple schemas account for a large fraction of the manually-written fixes found in open-source projects.

13. Since all fixing actions are comparatively simple, they do not affect the ranking of valid fixes, which is only based on suspiciousness of snapshots.

rank in the top 10 positions among the valid ones; hence AutoFix users only have to inspect the top fixes to decide with good confidence if any of them is deployable.

5 Experimental Evaluation

We performed an extensive experimental evaluation of the behavior and performance of AutoFix by applying it to over 200 faults found in various Eiffel programs. The experiments characterize the reproducible *average* behavior of AutoFix in a variety of conditions that are indicative of *general* usage. To ensure generalizable results, the evaluation follows stringent rules: the experimental protocol follows recommended guidelines [26] to achieve *statistically significant* results in the parts that involve randomization; the faults submitted to AutoFix come from four code bases of *different quality and maturity*; the experiments characterize usage with *limited computational resources*.

Two additional features distinguish this experimental evaluation from those of most related work (see Section 6). First, the experiments try to capture the usage of AutoFix as a fully automatic tool where user interaction is limited to selecting a project, pushing a button, and waiting for the results. The second feature of the evaluation is that it includes a detailed inspection of the quality of the automatically generated fixes, based on the distinction between *valid* and *proper* fixes introduced in Section 4.5.

5.1 Experimental Questions and Summary of Findings

Based on the high-level goals just presented, the experimental evaluation addresses the following questions:

- Q1 *How many* faults can AutoFix correct, and what are their characteristics?
- Q2 What is the *quality* of the fixes produced by AutoFix?
- Q3 What is the *cost* of fixing faults with AutoFix?
- Q4 How *robust* is AutoFix's performance in an "average" run?

The main findings of the evaluation are as follows:

- AutoFix produced valid fixes for 86 (or 42 percent) out of 204 randomly detected faults in various programs.
- Of the 86 valid fixes produced by AutoFix, 51 (or 59 percent) are proper, that is of quality comparable to those produced by professional programmers.
- AutoFix achieves its results with limited computational resources: AutoFix ran no more than 15 minutes per fault in 93.1 percent of the experiments; its median running time in all our experiments was 3 minutes, with a standard deviation of 6.3 minutes.
- AutoFix's behavior is, to a large extent, robust with respect to variations in the test cases produced by AutoTest: 48 (or 56 percent) of the faults that AutoFix managed to fix at least once were fixed (with possibly different fixes) in over 95 percent of the sessions. If we ignore the empty sessions where AutoTest did not manage to reproduce a fault, AutoFix produced a valid fix 41 percent of all non-empty sessions—when AutoFix is successful, it is *robustly* so.

TABLE 1
Size and Other Metrics of the Code Bases (the Dot Is the Decimal Mark; the Comma Is the Thousands Separator)

Code base	#C	#kLOC	#R	#Q	#Pre	#Post	#Inv
Base	11	26.548	1,504	169	1,147	1,270	209
TxtLib	10	12.126	780	48	97	134	11
Cards	32	20.553	1,479	81	157	586	58
ELearn	27	13.693	1,055	20	144	148	38
Total	**80**	**72.920**	**4,818**	**318**	**1,545**	**2,138**	**316**

5.2 Experimental Setup

All the experiments ran on the computing facilities of the Swiss National Supercomputing Centre consisting of Transtec Lynx CALLEO High-Performance Servers 2,840 with 12 physical cores and 48 GB of RAM. Each experiment session used exclusively one physical core at 1.6 GHz and 4 GB of RAM, whose computing power is similar to that of a commodity personal computer. Therefore, the experiments reflect the performance of AutoFix in a standard programming environment.

We now describe the code bases and the faults targeted by the experiments (Section 5.2.1), then present the experimental protocol (Section 5.2.2).

5.2.1 Experimental Subjects

The experiments targeted a total of 204 contract-violation faults collected from four code bases of different quality and maturity. The following discussion analyzes whether such a setup provides a sufficiently varied collection of subjects that exercise AutoFix in different conditions.

Code bases. The experiments targeted four code bases:

- `Base` is a data structure library. It consists of the standard data structure classes from the EiffelBase and Gobo projects, distributed with the EiffelStudio IDE and developed by professional programmers over many years.
- `TxtLib` is a library to manipulate text documents, developed at ETH Zurich by second-year bachelor's students with some programming experience.
- `Cards` is an on-line card gaming system, developed as project for DOSE, a distributed software engineering course organized by ETH [27] for master's students. Since this project is a collaborative effort involving groups in different countries, the students who developed `Cards` had heterogeneous, but generally limited, skills and experience with Eiffel programming and using contracts; their development process had to face the challenges of team distribution.
- `ELearn` is an application supporting e-learning, developed in another edition of DOSE.

Table 1 gives an idea of the complexity of the programs selected for the experiments, in terms of number of classes (#C), thousands of lines of code (#kLOC), number of routines (#R), Boolean queries (#Q), and number of contract clauses in preconditions (#Pre), postconditions (#Post), and class invariants (#Inv).

The data suggests that `Base` classes are significantly more complex than the classes in other code bases, but they

TABLE 2
Faults Used in the Fixing Experiments

Code base	#Faults	#Void	#Pre	#Post	#Inv	#Check	$\frac{\#F}{kLOC}$
Base	60	0	23	32	0	5	2.3
TxtLib	31	12	14	1	0	4	2.6
Cards	63	24	21	8	10	0	3.1
ELearn	50	16	23	8	3	0	3.7
Total	**204**	**52**	**81**	**49**	**13**	**9**	**2.8**

also offer a better interface with more Boolean queries that can be used by AutoFix (Section 3.3). The availability of *contracts* also varies significantly in the code bases, ranging from 0.76 precondition clauses per routine in Base down to only 0.11 precondition clauses per routine in Cards. This diversity in the quality of interfaces and contracts ensures that the experiments are representative of AutoFix's behavior in different conditions; in particular, they demonstrate the performance also with software of low quality and with very few contracts, where fault localization can be imprecise and unacceptable behavior may be incorrectly classified as passing for lack of precise oracles (thus making it more difficult to satisfactorily fix the bugs that are exposed by other contracts).

Faults targeted by the experiments. To select a collection of faults for our fixing experiments, we performed a preliminarily run of AutoTest [4] on the code bases and recorded information about all faults found that consisted of contract violations. These include violations of preconditions, postconditions, class invariants, and intermediate assertions (**check** instructions), but also violations of *implicit* contracts, such as dereferencing a void pointer and accessing an array element using an index that is out of bounds, and application-level memory and I/O errors such as a program terminating without closing an open file and buffer overruns. In contrast, we ignored lower-level errors such as disk failures or out-of-memory allocations, since these are only handled by the language runtime. Running AutoTest for two hours on each class in the code bases provided a total of 204 *unique* contract-violation faults (identified as discussed in Section 3.2). Table 2 counts these unique faults for each code base (#Faults), and also shows the breakdown into void-dereferencing faults (#Void), precondition violations (#Pre), postcondition violations (#Post), class invariant violations (#Inv), and check violations (#Check), as well as the number of faults per kLOC ($\frac{\#F}{kLOC}$). The figures in the last column give a rough estimate of the quality of the code bases, confirming the expectation that software developed by professional programmers adheres to higher quality standards.

The use of AutoTest for selecting faults has two principal consequences for this study:

- On the negative side, using AutoTest reduces the types of programs we can include in the experiments, as the random testing algorithm implemented in AutoTest has limited effectiveness with functionalities related to graphical user interfaces, networking, or persistence.
- On the positive side, using AutoTest guards against bias in the selection of faults in the testable classes, and makes the experiments representative of the primary intended usage of AutoFix: a completely automatic tool that can handle the whole debugging process autonomously.

To ensure that the faults found by AutoTest are "real", we asked, in related work [12], some of the maintainers of Base to inspect 10 faults, randomly selected among the 60 faults in Base used in our experiments; their analysis confirmed all of them as real bugs requiring to be fixed. Since Eiffel developers write both programs and their contracts, it is generally safe to assume that a contract violation exposes a genuine fault, since a discrepancy between implementation and specification must be reconciled somehow; this assumption was confirmed in all our previous work with AutoTest.

5.2.2 Experimental Protocol

The ultimate goal of the experiments is to determine the typical behavior of AutoFix in general usage conditions under constrained computational resources and a completely automatic process. Correspondingly, the experimental protocol involves a large number of repetitions, to ensure that the average results are statistically significant representatives of a typical run, and combines AutoTest and AutoFix sessions, to minimize the dependency of the quality of fixes produced by AutoFix on the choice of test cases, and to avoid requiring users to provide test cases.

For each unique fault **f** identified as in Section 5.2.1, we ran 30 AutoTest sessions of 60 minutes each, with the faulty routine as primary target. Each session produces a sequence of test cases generated at different times. Given a fault **f** in a routine r, we call *m-minute series on* **f** any prefix of a testing sequence generated by AutoTest on r. A series may include both passing and failing test cases. In our analysis we considered series of $m = 5$, 10, 15, 20, 30, 40, 50, and 60 minutes. The process determined 30 m-minute series (one per session) for every m and for every fault **f**; each such series consists of a set $T = P \cup F$ of passing P and failing F test cases.

Since the AutoFix algorithms are deterministic, an m-minute series on some fault **f** uniquely determines an AutoFix session using the tests in T to fix the fault **f**. The remainder of the discussion talks of *m-minute fixing session on* **f** to denote the unique AutoFix session run using some given m-minute series on **f**. In all, we recorded the fixes produced by 270 ($= 9 \times 30$) fixing sessions of various lengths on each fault; in each session, we analyzed at most 10 fixes—those ranked in the top 10 positions—and discarded the others (if any).

5.3 Experimental Results

The experimental data were analyzed through statistical techniques. Section 5.3.1 discusses how many valid fixes AutoFix produced in the experiments, and Section 5.3.2 how many of these were proper fixes. Section 5.3.3 presents the average AutoFix running times. Section 5.3.4 analyzes the performance of AutoFix over multiple sessions to assess its average behavior and its robustness.

5.3.1 How Many Faults AutoFix Can Fix

It is important to know for how many faults AutoFix managed to construct *valid* fixes in *some* of the repeated

TABLE 3
Number of Faults Fixed by AutoFix (*Valid* Fixes)

Code base	#Fixed	#Void	#Pre	#Post	#Inv	#Check
Base	26 (43%)	– (–)	18 (78%)	7 (22%)	– (–)	1 (20%)
TxtLib	14 (45%)	5 (42%)	5 (36%)	0 (0%)	– (–)	4 (100%)
Cards	31 (49%)	14 (58%)	13 (62%)	4 (50%)	0 (0%)	– (–)
ELearn	15 (30%)	4 (25%)	9 (39%)	2 (25%)	0 (0%)	– (–)
Total	**86 (42%)**	**23 (44%)**	**45 (56%)**	**13 (27%)**	**0 (0%)**	**5 (56%)**

TABLE 4
Types of Faults That AutoFix Could Not Fix

Code base	#NotFixed	#NoFail	#Complex	#Contract	#Design
Base	34	3	8	10	13
TxtLib	17	1	5	10	1
Cards	32	6	4	16	6
ELearn	35	0	13	14	8
Total	**118**	**10**	**30**	**50**	**28**

experiments. The related questions of whether these results are sensitive to the testing time or depend on chance are discussed in the following sections.

When AutoFix succeeds. The second column of Table 3 lists the total number of unique faults for which AutoFix was able to build a *valid* fix and *rank* it among the top 10 during *at least one* of the 55,080 (270 sessions for each of the 204 unique faults) fixing sessions, and the percentage relative to the total number of unique faults in each code base. The other columns give the breakdown into the same categories of fault as in Table 2. Overall, AutoFix succeeded in fixing 86 (or 42 percent) of the faults. Section 5.3.4 discusses related measures of *success rate*, that is the percentage of sessions that produced a valid fix.

The fixing process is in general non-monotonic; that is, there are faults f on which there exists some successful m-minute fixing session but no successful n-minute fixing sessions for some $n > m$. The reason is the randomness of AutoTest: a short AutoTest run may produce better, if fewer, tests for fixing than a longer run, which would have more chances of generating spurious or redundant passing tests. Non-monotonic behavior is, however, very infrequent: we observed it only for two faults (one in `Base` and one in `Cards`) which were overly sensitive to the kinds of test cases generated. In both cases, the faults were fixed in all sessions but those corresponding to a single intermediate testing time (respectively, 15 and 20 minutes). This corroborates the idea that non-monotonicity is an ephemeral effect of randomness of test-case generation, and suggests that it is not a significant issue in practice.

When AutoFix fails. To understand the limitations of our technique, we manually analyzed all the faults for which AutoFix always failed, and identified four scenarios that prevent success. Table 4 lists the number of faults not fixed (column #NotFixed) and the breakdown into the scenarios described next.

Faults hard to reproduce. A small portion of the faults identified during the preliminary 2-hour sessions (Section 5.2.1) could not be reproduced during the shorter AutoTest sessions used to provide input to AutoFix (Section 5.2.2). Without failing test cases[14] the AutoFix algorithms cannot possibly be expected to work. Column #NoFail in Table 4 lists the faults that we could not reproduce, and hence could not fix, in the experiments.[15]

Complex patches required. While a significant fraction of fixes are simple [24], some faults require complex changes to the implementation (for example, adding a loop or handling special cases differently). Such patches are currently out of the scope of AutoFix; column #Complex of Table 4 lists the faults that would require complex patches.

Incorrect or incomplete contracts. AutoFix assumes contracts are correct and tries to fix implementations based on them. In practice, however, contracts contain errors too; in such cases, AutoFix may be unable to satisfy an incorrect specification with changes to the code. A related problem occurs when contracts are missing some constraints—for example about the invocation order of routines—that are documented informally in the comments; faults generated by violating such informally-stated requisites are spurious, and AutoFix's attempts thus become vain. Column #Contract of Table 4 lists the faults involving incorrect or incomplete contracts that AutoFix cannot fix. (In recent work [28], we developed a fixing technique that suggests changes to incorrect or inconsistent contracts to *remove* faults.)

Design flaws. The design of a piece of software may include inconsistencies and dependencies between components; as a consequence fixing some faults may require changing elements of the design—something currently beyond what AutoFix can do. The design flaws that AutoFix cannot correct often involve inheritance; for example, a class *LINKED_SET* in `Base` inherits from *LINKED_LIST* but does not uniformly changes its contracts to reflect the fact that a set does not have duplicates while a list may. Fixing errors such as this requires a substantial makeover of the inheritance hierarchy, of the interfaces, or both. Column #Design of Table 4 lists the faults due to design flaws that AutoFix cannot fix.

Which fix schemas are used. Not all four schemas available to AutoFix (Section 4.4.1) are as successful at generating valid fixes. Table 5 shows the number of faults successfully fixed using each of the schemas *a*, *b*, *c*, and *d* in Fig. 6. For reference, column #F shows the total number of faults in each code base; since two valid fixes for the same fault may use different schemas, the total number of faults fixed with any schema is larger than the numbers in column #F.

TABLE 5
Number of Faults Fixed Using Each of the Fix Schemas in Fig. 6

Code base	#F	Schema (a)	Schema (b)	Schema (c)	Schema (d)
Base	26	9	18	18	23
TxtLib	14	0	12	0	6
Cards	31	0	27	6	25
ELearn	15	0	11	4	11
Total	**86**	**9**	**68**	**28**	**65**

14. As a side remark, AutoFix managed to fix 19 faults for which AutoTest could generate *only failing* tests; seven of those fixes are even proper.

15. Even if AutoTest were given enough time to generate failing tests, AutoFix would still not succeed on these faults due to complex patch required (four faults) or incorrect contracts (six faults).

TABLE 6
Number of Faults Fixed by AutoFix (*Proper* Fixes)

Code base	#Fixed	#Void	#Pre	#Post	#Inv	#Check
Base	**12 (20%)**	– (–)	12 (52%)	0 (0%)	– (–)	0 (0%)
TxtLib	**9 (29%)**	4 (33%)	2 (14%)	0 (0%)	– (–)	3 (75%)
Cards	**18 (29%)**	10 (42%)	8 (38%)	0 (0%)	0 (0%)	– (–)
ELearn	**12 (24%)**	3 (19%)	7 (30%)	2 (25%)	0 (0%)	– (–)
Total	**51 (25%)**	**17 (33%)**	**29 (36%)**	**2 (4%)**	**0 (0%)**	**3 (33%)**

TABLE 7
Number of Faults with Proper Fixes Using Each of the Fix Schemas in Fig. 6

Code base	#F	Schema (a)	Schema (b)	Schema (c)	Schema (d)
Base	12	0	7	5	7
TxtLib	9	0	8	0	0
Cards	18	0	18	0	3
ELearn	12	0	7	4	3
Total	**51**	**0**	**40**	**9**	**13**

Schemas *b* and *d* are the most successful ones, producing valid fixes for 79 and 75 percent of the 86 fixable faults; together, they can fix *all* the 86 faults. This means that the most effectively deployable fixing strategies are: "execute a repair action when a suspicious state holds" (schema *b*); and "execute an alternative action when a suspicious state holds, and proceed normally otherwise" (schema *d*).

In our experiments, AutoFix produced valid fixes for 86 (42%) of 204 faults.

5.3.2 Quality of Fixes

What is the quality of the valid fixes produced by AutoFix in our experiments? We manually inspected the valid fixes and determined how many of them can be considered *proper*, that is genuine corrections that *remove* the root of the error (see Section 4.5).

Since what constitutes correct behavior might be controversial in some corner cases, we tried to leverage as much information as possible to determine the likely intent of developers, using comments, inspecting client code, and consulting external documentation when available. In other words, we tried to classify a valid fix as proper only if it really meets the expectations of real programmers familiar with the code base under analysis. Whenever the notion of proper was still undetermined, we tried to be conservative as much as possible. While we cannot guarantee that the classification is indisputable, we are confident it is overall very reasonable and sets high standards of quality.

The second column of Table 6 lists the total number of unique faults for which AutoFix was able to build a *proper* fix and *rank* it among the top 10 during *at least one* of the fixing sessions, and the percentage relative to the total number of faults in code base. The other columns give the breakdown into the same categories of fault as in Tables 2 and 3. Overall, AutoFix produces proper fixes in the majority (59 percent of 86 faults) of cases where it succeeds, corresponding to 25 percent of all unique faults considered in the experiments; these figures suggest that the quality of fixes produced by AutoFix is often high.

The quality bar for proper fixes is set quite high: many valid but non-proper fixes could still be usefully deployed, as they provide effective work-arounds that can at least avoid system crashes and allow executions to continue. Indeed, this kind of "first-aid" patches is the primary target of related approaches described in Section 6.3.

We did not analyze the ranking of proper fixes within the top 10 valid fixes reported by AutoFix. The ranking criteria (Section 4.6) are currently not precise enough to guarantee that proper fixes consistently rank higher than improper ones. Even if the schemas used by AutoFix lead to textually simple fixes, analyzing up to 10 fixes may introduce a significant overhead; nonetheless, especially for programmers familiar with the code bases,[16] the time spent analyzing fixes is still likely to trade off favorably against the effort that would be required by a manual debugging process that starts from a single failing test case. Future work will empirically investigate the human effort required to evaluate and deploy fixes produced by AutoFix.

Which fix schemas are used. The effectiveness of the various fix schemas becomes less evenly distributed when we look at proper fixes. Table 7 shows the number of faults with a proper fix using each of the schemas *a*, *b*, *c*, and *d* in Fig. 6; it is the counterpart of Table 5 for proper fixes. schema *a* is used in no proper fix, whereas schema *b* is successful with 78 percent of the 51 faults for which AutoFix generates a proper fix; schemas *b* and *d* together can fix 44 out of those 51 faults. These figures demonstrate that unconditional fixes (schema *a*) were not useful for the faults in our experiments. Related empirical research on manually-written fixes [29] suggests, however, that there is a significant fraction of faults whose natural corrections consist of unconditionally adding an instruction; this indicates that schema *a* may still turn out to be applicable to code bases other than those used in our experiments (or that AutoFix's fault localization based on Boolean conditions in snapshots naturally leads to conditional fixes).

In our experiments, AutoFix produced ***proper*** *fixes (of quality comparable to programmer-written fixes) for 51 (25%) of 204 faults.*

5.3.3 Time Cost of Fixing

Two sets of measures quantify the cost of AutoFix in terms of running time. The first one is the average running time for AutoFix alone; the second one is the average total running time per fix produced, including both testing and fixing.

Fixing time per fault. Fig. 7 shows the distribution of running times for AutoFix (independent of the length of the preliminary AutoTest sessions) in all the experiments.[17] A bar at position x whose black component reaches height y_B, gray component reaches height $y_G \geq y_B$, and white component reaches height $y_W \geq y_G$ denotes that y_W fixing sessions

16. During the data collection phase for this paper, it took the first author 3 to 6 minutes to understand and assess each valid fix for a given fault.

17. AutoFix ran with a timeout of 60 minutes, which was reached only for two faults.

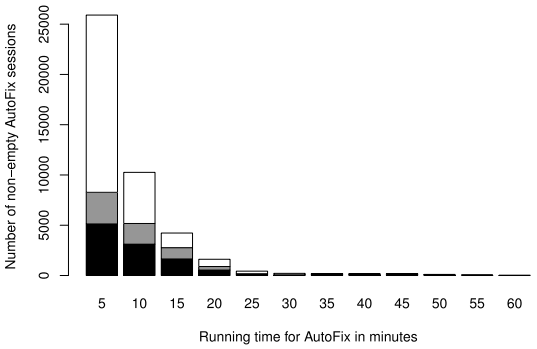


Fig. 7. Distribution of running times for AutoFix, independent of the length of the preliminary AutoTest sessions (black bars: sessions with proper fixes; gray bars: sessions with valid fixes; white bars: all sessions).

terminated in a time between $x - 5$ and x minutes; y_G of them produced a valid fix; and y_B of them produced a proper fix. The pictured data does not include the 11,670 "empty" sessions where AutoTest failed to supply any failing test cases, which terminated immediately without producing any fix. The distribution is visibly skewed towards shorter running times, which demonstrates that AutoFix requires limited amounts of time in general.

Table 8 presents the same data about non-empty fixing sessions in a different form: for each amount of AutoFix running time (first column), it displays the number and percentage of sessions that terminated in that amount of time (#Sessions), the number and percentage of those that produced a valid fix (#Valid), and the number and percentage of those that produced a proper fix (#Proper). Table 9 shows the minimum, maximum, mean, median, standard deviation, and skewness of the running times (in minutes) across: all fixing sessions, all non-empty sessions, all sessions that produced a valid fix, and all sessions that produced a proper fix.

Total time per fix. The *total* running time of a fixing session also depends on the time spent generating input test cases; the session will then produce a variable number of valid fixes ranging between zero and 10 (remember that we ignore fixes not ranked within the top 10). To have a finer-grained measure of the running time based on these factors, we define the *unit fixing time* of a combined session that runs AutoTest for t_1 and AutoFix for t_2 and produces $v > 0$ valid fixes as $(t_1 + t_2)/v$. Fig. 8a shows the distribution of unit fixing times in the experiments: a bar at position x reaching height y denotes that y sessions produced at least one valid fix each, spending an average of x minutes of testing and fixing on each. The distribution is strongly skewed towards short fixing times, showing that the vast majority of valid fixes is produced in 15 minutes or less. Table 10 shows the statistics of unit fixing times for all sessions producing valid fixes, and for all sessions producing proper fixes. Fig. 8b shows the same distribution of unit fixing times as Fig. 8a but for proper fixes. This distribution is also skewed towards shorted fixing times, but much less so than the one in Fig. 8a: while the majority of valid fixes can be produced in 35 minutes or less, proper fixes require more time on average, and there is a substantial fraction of proper fixes requiring longer times up to about 70 minutes.

TABLE 8
Distribution of Running Times for AutoFix

min. Fixing	#Sessions	#Valid	#Proper
5	25905 (59.7%)	8275 (31.9%)	5130 (19.8%)
10	36164 (83.4%)	13449 (37.2%)	8246 (22.8%)
15	40388 (93.1%)	16220 (40.2%)	9892 (24.5%)
20	42003 (96.9%)	17114 (40.7%)	10432 (24.8%)
25	42436 (97.9%)	17295 (40.8%)	10543 (24.8%)
30	42650 (98.4%)	17371 (40.7%)	10607 (24.9%)
40	43025 (99.2%)	17670 (41.1%)	10799 (25.1%)
50	43318 (99.9%)	17918 (41.4%)	11013 (25.4%)
60	43365 (100.0%)	17954 (41.4%)	11046 (25.5%)

TABLE 9
AutoFix Running Time Statistics (Times are in Minutes)

	min	max	mean	median	stddev	skew
All	0.0	60	4.8	3.0	6.3	3.2
Non-empty	0.0	60	6.1	4.0	6.5	3.2
Valid	0.5	60	7.8	5.5	7.6	2.8
Proper	0.5	60	8.1	5.4	8.3	2.9

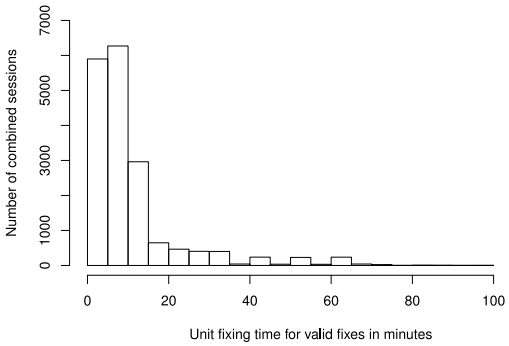
The unit fixing time is undefined for sessions producing no fixes, but we can still account for the time spent by fruitless fixing sessions by defining the *average unit fixing time* of a group of sessions as the total time spent testing and fixing divided by the total number of valid fixes produced (assuming we get at least one valid fix). Table 11 shows, for each choice of testing time, the average unit fixing time for valid fixes (second column) and for proper fixes (third column); the last line reports the average unit fixing time over all sessions: 19.9 minutes for valid fixes and 74.2 minutes for proper fixes.

Looking at the big picture, the fixing times are prevalently of moderate magnitude, suggesting that AutoFix (and its usage in combination with AutoTest) can make an efficient usage of computational time and quickly produce useful results in most cases. The experimental results also suggest practical guidelines to use AutoFix and AutoTest: as a rule of thumb, running AutoTest for five to ten minutes has a fair chance of producing test cases for AutoFix to correct an "average" fault.

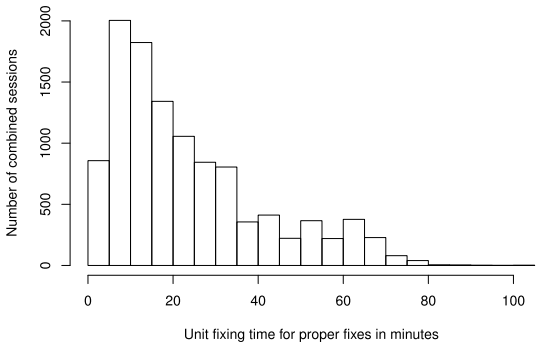
In our experiments, AutoFix took on average less than 20 minutes per valid fix, including the time required to generate suitable tests with AutoTest.

5.3.4 Robustness

The last part of the evaluation analyzes the robustness and repeatability of AutoFix sessions. The AutoFix algorithm is purely deterministic, given as input an annotated program and a set of passing and failing test cases exposing a fault in the program. In our experiments, however, all the tests come from AutoTest, which operates a randomized algorithm, so that different runs of AutoTest may produce test suites of different quality. We want to assess the robustness of AutoFix with respect to different choices of input test suites, that is how AutoFix's output depends on the test cases supplied. Assessing robustness is important to demonstrate that our evaluation is indicative of *average* usage,



(a) Distribution of unit fixing times for *valid* fixes.



(b) Distribution of unit fixing times for *proper* fixes.

Fig. 8. Distribution of unit fixing times, including the time spent in the preliminary AutoTest sessions.

and its results do not hinge on having used a particularly fortunate selection of tests.

Our experiments consisted of many repeated runs of AutoTest, each followed by AutoFix runs using the generated test as input. To assess robustness we fix the testing time, and we measure the percentage of AutoFix runs, on each of the repeated testing sessions terminating within the allotted testing time, that produced a valid fix. A high percentage shows that AutoFix was successful in most of the repeated testing runs, and hence largely independent of the specific performance of AutoTest; to put it differently, a random testing session followed by a fixing sessions has a high chance of producing a valid fix.

Formally, to measure the robustness with respect to choice of test cases, we introduce the notion of *success rate*: given a fault $\mathbf{f}$ and a testing time m, the *m-minute absolute success rate on* $\mathbf{f}$ is defined as the percentage of m-minute fixing sessions on $\mathbf{f}$ that produce at least one valid fix; the *relative success rate* is defined similarly but the percentage is relative only to non-empty fixing sessions (where AutoTest produced at least one failing test case). Fig. 9 shows the distribution of the absolute (Fig. 9a) and relative (Fig. 9b) success rates for all "fixable" faults—for which AutoFix produced a valid fix at least once in our experiments—for any testing time m. The graphs demonstrate that AutoFix has repeatable behavior with a strong majority of faults, largely insensitive to the specific input test cases. The relative success rates, in particular, exclude the empty AutoTest sessions (which are concentrated on some "hard to reproduce faults" as discussed in Section 5.3.1) and thus characterize the robustness of AutoFix's behavior on the "approachable" faults. (The fact that a classification into "approachable" and "hard" faults for AutoFix naturally emerges further indicates that the kinds of faults used in this evaluation are varied.)

To have a quantitative look at the same data, Table 12 displays, for each testing time m, the number of faults that were fixed successfully—producing a valid fix—in at least $X\%$ of the m-minute fixing sessions, for percentages $X = 50, 80, 90, 95$.[18] Each table entry also shows, in parentheses, the percentage of the fixed faults, relative to the 86 fixable faults that AutoFix fixed at least once; the data is shown for both the relative and the absolute success rate. For example, AutoFix was successful at least 95 percent of the times with 56 percent of all fixable faults; or even with 79 percent of all fixable faults provided with at least one failing test case. The last line displays the statistics over all testing sessions of any length. The aggregated data over all fixing sessions for all faults is the following: 32 percent of all sessions and 41 percent of all non-empty sessions produced a valid fix. These success rates suggest a high repeatability of fixing.

Fig. 10 and Table 13 display similar data about successful sessions that produced at least one *proper* fix, with percentages relative to all faults for which AutoFix produced a proper fix at least once in our experiments. The aggregated data over all fixing sessions for all faults is the following: 20 percent of all sessions and 25 percent of all non-empty sessions produced a proper fix; these percentages are quite close to the 25 percent of all faults for which AutoFix produces at least once a proper fix (Table 6). The data for proper fixes is overall quite similar to the one for valid fixes. The absolute figures are a bit smaller, given that the requirement of proper fixes is more demanding, but still support the hypothesis that AutoFix's behavior is often robust and largely independent of the quality of provided test cases.

In our experiments, AutoFix produced valid fixes in 41% of the sessions with valid input tests.

TABLE 10
Unit Fixing Times Statistics (Times Are in Minutes and Include the Time Spent in the Preliminary AutoTest Sessions)

	min	max	mean	median	stddev	skew
Valid	0.7	98.6	10.8	6.9	12.1	2.9
Proper	1.0	101.1	23.5	17.9	17.9	1.1

5.4 Limitations and Threats to Validity

Limitations. AutoFix relies on a few assumptions, which may restrict its practical applicability.

18. All else being equal, the number of fixed faults is larger when considering *relative* success rates: a relative success rate of $X\% = r/n$ corresponds to r successful fixing sessions out of n non-empty sessions; an absolute success rate of $X\% = a/(n+e)$ for the same testing time corresponds to a successful fixing sessions out of n non-empty sessions and e empty sessions; since $r/n = a/(n+e)$ and $e \geq 0$, it must be $r \geq a$; hence the number of unique faults is also larger in general for the relative rate.

TABLE 11
Average Unit Fixing Times for Different Testing Times (Times Are in Minutes)

min. Testing	min. Valid	min. Proper
5	6.0	22.0
10	8.9	32.5
15	11.9	43.7
20	14.6	54.0
25	17.7	65.3
30	20.4	76.7
40	26.1	97.3
50	31.9	121.6
60	37.3	143.5
All	**19.9**	**74.2**

Contracts or a similar form of annotation must be available in the source code. The simple contracts that programmers write [7] are sufficient for AutoFix; and having to write contracts can be traded off against not having to write test cases. Requiring contracts does not limit the applicability of our technique to Eiffel, given the increasing availability of support for contracts in mainstream programming languages. However, the software projects that use contracts in their development is still a small minority [7], which restricts broader applicability of AutoFix on the software that is currently available without additional annotation effort.

Whether writing contracts is a practice that can become part of mainstream software development is a long-standing question. Our previous experience is certainly encouraging, in that using contracts does not require highly-trained programmers, and involves efforts that can be traded off against other costs (e.g., maintenance [30]) and are comparable to those required by other more accepted practices. For example, EiffelBase's contracts-to-code ratio is around 0.2 [12]; while detailed quantitative data about industrial experiences with a more accepted practice such as test-driven development is scarce, the few references that indicate quantitative measures [31], [32], [33] report test-LOC-to-application-LOC ratios between 0.4 and 1.0 for projects of size comparable to EiffelBase. More extensive assessments belong to future work beyond the scope of the present paper.

Functional faults are the primary target of AutoFix, given that contracts provide an effective specification of functional correctness. This excludes, for example, violation of liveness properties (e.g., termination) or low-level I/O runtime errors (Section 5.2.1). Nonetheless, the expressiveness of contracts is significant, and in fact we could identify various categories of contract-violation faults that AutoFix can or cannot fix (Section 5.3.1).

Correctness of contracts is assumed by AutoTest, which uses them as oracles, and by AutoFix, which fixes implementations accordingly. Since contracts have errors too, this may affect the behavior of AutoFix on certain faults (see Section 5.3.1). Anyway, the line for correctness must be drawn somewhere: test cases may also include incorrect usages or be incorrectly classified.

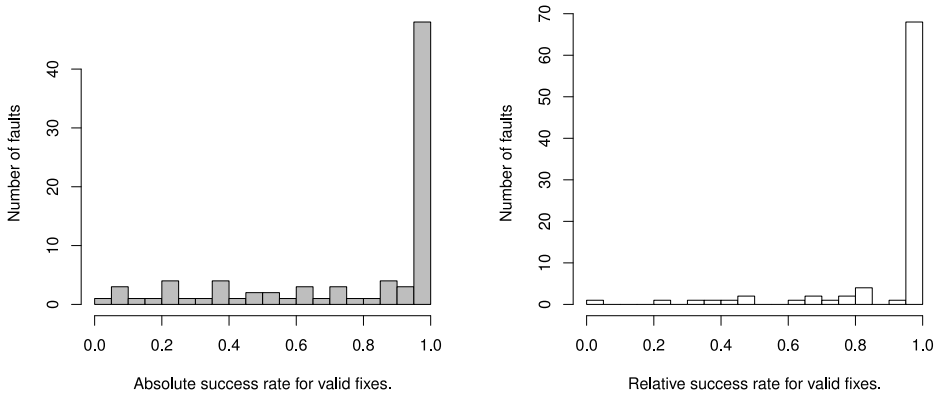
Types of fixes generated by AutoFix include only a subset of all possible actions (Section 4.3) and are limited to simple schema (Section 4.4). This limits the range of fixes that AutoFix can generate; at the same time, it helps reduce the search space of potential fixes, focusing on the few schema that cover the majority of cases [24], [25].

Threats to validity. While we designed the evaluation of AutoFix targeting a broad scope and repeatable results, a few threats to generalizability remain.

Automatically generated test cases were used in all our experiments. This provides complete automation to the debugging process, but it also somewhat restricts the kinds of projects and the kinds of faults that we can try to those that we can test with AutoTest. We plan to experiment with manually-written test cases in future work.

Unit tests were used in all our experiments, as opposed to system tests. Unit tests are normally smaller, which helps with fault localization and, consequently, to reduce the search space of possible fixes. The fact that unit tests are produced as part of fairly widespread practices such as test-driven development [31] reflects positively on the likelihood that they be available for automated fixing.

Size and other characteristics (type of program, programming style, and so on) of the programs used in the evaluation were constrained by the fundamental choice of targeting object-oriented programs using contracts that can be tested with AutoTest. This implies that further experiments are needed to determine to what extent the algorithms used by AutoFix scale to much larger code bases—possibly with large-size modules and system-wide executions—and which design choices should be reconsidered in that context. To partly mitigate this threat to generalizability, we selected experimental subjects of non-trivial size



(a) Absolute success rate. (b) Relative success rate.

Fig. 9. Distribution of success rates for *valid* fixes.

TABLE 12
Repeatability of AutoFix on Faults That Produced Some *Valid* Fixes

Success rate:	**50%**		**80%**		**90%**		**95%**	
min. Testing	*relative*	*absolute*	*relative*	*absolute*	*relative*	*absolute*	*relative*	*absolute*
5	83 (97%)	58 (67%)	80 (93%)	49 (57%)	78 (91%)	46 (53%)	75 (87%)	40 (47%)
10	83 (97%)	62 (72%)	77 (90%)	56 (65%)	75 (87%)	51 (59%)	69 (80%)	45 (52%)
15	81 (94%)	65 (76%)	76 (88%)	58 (67%)	71 (83%)	52 (60%)	68 (79%)	48 (56%)
20	82 (95%)	68 (79%)	76 (88%)	58 (67%)	70 (81%)	54 (63%)	67 (78%)	51 (59%)
25	80 (93%)	68 (79%)	72 (84%)	58 (67%)	70 (81%)	56 (65%)	65 (76%)	51 (59%)
30	81 (94%)	69 (80%)	74 (86%)	59 (69%)	70 (81%)	56 (65%)	68 (79%)	53 (62%)
40	79 (92%)	69 (80%)	71 (83%)	61 (71%)	68 (79%)	58 (67%)	65 (76%)	55 (64%)
50	79 (92%)	70 (81%)	73 (85%)	62 (72%)	69 (80%)	59 (69%)	63 (73%)	53 (62%)
60	78 (91%)	71 (83%)	73 (85%)	61 (71%)	68 (79%)	59 (69%)	67 (78%)	57 (66%)
All	**79 (92%)**	**67 (78%)**	**73 (85%)**	**56 (65%)**	**69 (80%)**	**51 (59%)**	**68 (79%)**	**48 (56%)**

exhibiting variety in terms of quality, maturity, and available contracts—within the constraints imposed by our fundamental design choices, as discussed in Section 5.2.1.

Variability of performance relative to different choices for the various heuristics used by AutoFix has not been exhaustively investigated. While most heuristics rely on well-defined notions, and we provided the rationale for the various design choices, there are a few parameters (such as α, β, and γ in Section 4.2.2) whose impact we have not investigated as thoroughly as other aspects of the AutoFix algorithm. As also discussed in Section 4.2.2, the overall principles behind the various heuristics are not affected by specific choices for these parameters; therefore, the impact of this threat to generalizability is arguably limited.

Limited computational resources were used in all our experiments; this is in contrast to other evaluations of fixing techniques [34]. Our motivation for this choice is that we conceived AutoFix as a tool integrated within a personal development environment, usable by individual programmers in their everyday activity. While using a different approach to automatic fixing could take advantage of massive computational resources, AutoFix was designed to be inexpensive and evaluated against this yardstick.

Classification of fixes into proper and improper was done manually by the first author. While this may have introduced a classification bias, it also ensured that the classification was done by someone familiar with the code bases, and hence in a good position to understand the global effects of suggested fixes. Future work will investigate this issue empirically, as done in recent related work [35].

Programmer-written contracts were used in all our experiments. This ensures that AutoFix works with the kinds of contracts that programmers tend to write. However, as future work, it will be interesting to experiment with stronger higher-quality contracts to see how AutoFix performance is affected. In recent work [12] we obtained good results with this approach applied to testing with AutoTest.

6 Related Work on Automated Fixing

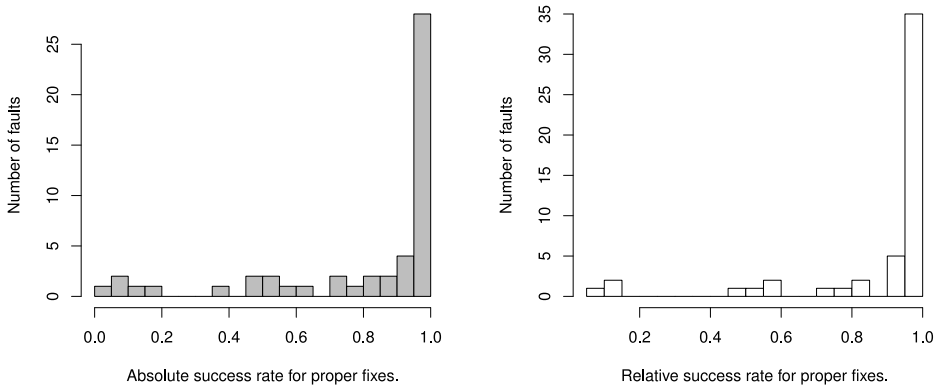
We present the related work on automated program fixing in three areas: techniques working on the source code (as AutoFix does); applications to specialized domains; and techniques that operate dynamically at runtime.

6.1 Source-Code Repairs

Techniques such as AutoFix target the source code to permanently *remove* the buggy behavior from a program.

Machine-learning techniques. Machine-learning techniques can help search the space of candidate fixes efficiently and support heuristics to scale to large code bases.

Jeffrey et al. [36] present BugFix, a tool that summarizes existing fixes in the form of *association rules*. BugFix then tries to apply existing association rules to new bugs. The user can also provide feedback—in the form of new fixes or validations of fixes provided by the algorithm—thus ameliorating the performance of the algorithm over time.



(a) Absolute success rate. (b) Relative success rate.

Fig. 10. Distribution of success rates for *proper* fixes.

TABLE 13
Repeatability of AutoFix on Faults That Produced Some *Proper* Fixes

Success rate:	50%		80%		90%		95%	
min. Testing	*relative*	*absolute*	*relative*	*absolute*	*relative*	*absolute*	*relative*	*absolute*
5	45 (88%)	35 (69%)	42 (82%)	31 (61%)	41 (80%)	41 (80%)	39 (76%)	24 (47%)
10	47 (92%)	41 (80%)	43 (84%)	35 (69%)	42 (82%)	42 (82%)	36 (71%)	27 (53%)
15	47 (92%)	41 (80%)	43 (84%)	37 (73%)	39 (76%)	39 (76%)	36 (71%)	29 (57%)
20	47 (92%)	43 (84%)	43 (84%)	37 (73%)	40 (78%)	40 (78%)	35 (69%)	27 (53%)
25	48 (94%)	44 (86%)	42 (82%)	37 (73%)	39 (76%)	39 (76%)	34 (67%)	28 (55%)
30	46 (90%)	43 (84%)	42 (82%)	37 (73%)	41 (80%)	41 (80%)	39 (76%)	32 (63%)
40	47 (92%)	45 (88%)	41 (80%)	39 (76%)	39 (76%)	39 (76%)	34 (67%)	32 (63%)
50	47 (92%)	45 (88%)	42 (82%)	39 (76%)	39 (76%)	39 (76%)	33 (65%)	31 (61%)
60	47 (92%)	45 (88%)	41 (80%)	39 (76%)	40 (78%)	40 (78%)	34 (67%)	31 (61%)
All	**47 (92%)**	**43 (84%)**	**42 (82%)**	**36 (71%)**	**40 (78%)**	**40 (78%)**	**35 (69%)**	**28 (55%)**

Other authors applied *genetic algorithms* to generate suitable fixes. Arcuri and Yao [37], [38] use a co-evolutionary algorithm where an initially faulty program and some test cases compete to evolve the program into one that satisfies its formal specification.

Weimer et al. [39], [40] describe GenProg, a technique that uses genetic programming[19] to mutate a faulty program into one that passes all given test cases. GenProg has been extensively evaluated [34], [41] with various open-source programs, showing that it provides a scalable technique, which can produce non-trivial corrections of subtle bugs, and which works without any user annotations (but it requires a regression test suite).

Kim et al. [35] describe Par, a technique that combines GenProg's genetic programming with a rich predefined set of fix patterns (suggested by human-written patches). Most of the fix patterns supported by Par are covered by AutoFix's synthesis strategies (Section 4.3); the few differences concern the usage of overloaded methods—a feature not available in the Eiffel language, and hence not covered by AutoFix. Par has also been extensively evaluated, with a focus on *acceptability* of patches: the programmers involved in the study tended to consider the patches generated by Par more acceptable than those generated by GenProg, and often as acceptable as human-written patches for the same bugs. The notion of acceptability addresses similar concerns to our notion of proper fix, since they both capture quality as perceived by human programmers beyond the objective yet weak notion of validity, although the two are not directly comparable.

Of the several approaches to source-code general-purpose program repair discussed in this section, GenProg and Par are the only ones that have undergone evaluations comparable with AutoFix's: the other approaches have only been applied to seeded faults [38], [42], [43], to few benchmarks used for fault localization [36], or do not aim at complete automation [44].

GenProg can fix 52 percent of 105 bugs with the latest improvements [34]; Par fixes 23 percent of 119 bugs (GenProg fixes 13 percent of the same 119 bugs [35]). In our experiments in Section 5, we target almost twice as many bugs (204) and AutoFix fixes 42 percent of them. Whereas these quantitative results should not directly be compared because they involve different techniques and faults, they demonstrate that all three approaches produce interesting results and have been thoroughly evaluated. GenProg's and Par's evaluations have demonstrated their scalability to large programs: GenProg worked on eight C programs totaling over five million lines of code; Par worked on six Java programs totaling nearly 500 thousand lines of code. AutoFix's evaluation targeted a total of 72 thousand lines of Eiffel code; while lines of code is a coarse-grained measure of effort, more experiments are needed to conclusively evaluate AutoFix's scalability on much larger programs. The test cases used in GenProg's and Par's evaluations (respectively, around 10 thousand and 25 thousand) do not seem to be directly comparable with those used by AutoFix: GenProg and Par use manually-written tests, which may include system tests as well as unit tests; AutoFix does not require user-written test cases (and uses fewer on average anyway) but uses automatically generated tests that normally exercise only a limited subset of the instructions in the whole program. The sensitivity of GenProg or Par about the input test suite have not been systematically investigated,[20] and therefore we do not know if they could perform well with tests generated automatically. In contrast, our experiments show that AutoFix is robust with respect to the input tests, and in fact it works consistently well with tests randomly generated given the simple contracts available in Eiffel programs. Another advantage of leveraging contracts is that AutoFix can naturally target *functional* errors (such as those shown in Section 2).

Weimer et al.'s evaluation of fix *quality* has been carried out only for a sample of the bugs, and mostly in terms of induced runtime performance [41]. It is therefore hard to compare with AutoFix's. Finally, AutoFix works with remarkably limited computational resources: using the same pricing scheme used in GenProg's evaluation [34],[21] AutoFix would require a mere \$0.01 per valid fix (computed as $0.184 \times \text{total fixing time in hours} / \text{total number of valid fixes}$) and \$0.03 per proper fix; or \$0.06 per valid and \$0.23 per proper fix including

19. See also Arcuri and Briand's remarks [26], Section 2] on the role of evolutionary search in Weimer et al.'s experiments [39].

20. GenProg's sensitivity to the design choices of its genetic algorithm has been recently investigated [45].

21. We consider *on-demand instances* of Amazon's EC2 cloud computing infrastructure, costing \$0.184 per wall-clock hour at the time of GenProg's experiments.

the time to generate tests—two orders of magnitude less than GenProg's $7.32 per valid fix.

Axiomatic reasoning. He and Gupta [42] present a technique that compares two program states at a faulty location in the program. The comparison between the two program states illustrates the source of the error; a change to the program that reconciles the two states fixes the bug. Unlike our work, theirs compares states purely statically with modular weakest precondition reasoning. A disadvantage of this approach is that modular weakest precondition reasoning may require detailed postconditions (typically, full functional specifications in first-order logic) in the presence of routine calls: the effects of a call to *foo* within routine *bar* are limited to what *foo's* postcondition specifies, which may be insufficient to reason about *bar's* behavior. Even if the static analysis were done globally instead of modularly, it would still require detailed annotations to reason about calls to native routines, whose source code is not available. This may limit the applicability to small or simpler programs; AutoFix, in contrast, compares program states mostly dynamically, handling native calls and requiring only simple annotations for postconditions. Another limitation of He and Gupta's work is that it builds fix actions by *syntactically* comparing the two program states; this restricts the fixes that can be automatically generated to changes in expressions (for example, in off-by-one errors). AutoFix uses instead a combination of heuristics and fix schemas, which makes for a flexible usage of a class's public routines without making the search space of possible solutions intractably large.

Constraint-based techniques. Gopinath et al. [43] present a framework that repairs errors due to value misuses in Java programs annotated with pre- and postconditions. A repairing process with the framework involves encoding programs as relational formulae, where some of the values used in "suspicious" statements are replaced by free variables. The conjunction of the formula representing a program with its pre- and postcondition is fed to a SAT solver, which suggests suitable instantiations for the free variables. The overall framework assumes an external fault localization scheme to provide a list of suspicious statements; if the localization does not select the proper statements, the repair will fail. Solutions using dynamic analysis, such as AutoFix, have a greater flexibility in this respect, because they can better integrate fault localization techniques—which are also typically based on dynamic analysis. As part of future work, however, we will investigate including SAT-based techniques within AutoFix.

Nguyen et al. [46] build on previous work [47] about detecting suspicious expressions to automatically synthesize possible replacements for such expression; their SemFix technique replaces or adds constants, variables, and operators to faulty expressions until all previously failing tests become passing. The major differences with respect to AutoFix are that SemFix's fault localization is based on statements rather than snapshots, which gives a coarser granularity; and that the fixes produced by SemFix are restricted to changes of right-hand sides of assignments and Boolean conditionals, whereas AutoFix supports routine calls, more complex expression substitutions, and conditional schemas. This implies that AutoFix can produce fixes that are cumbersome or impossible to build using SemFix. For example, conditional fixes are very often used by AutoFix (Tables 5 and 7) but can be generated by SemFix only if a conditional already exists at the repair location; and supporting routine calls in fixes takes advantage of modules with a well-designed API.

Model-driven techniques. Some automated fixing methods exploit finite-state abstractions to detect errors or to build patches. AutoFix also uses a form of finite-state abstraction as one way to synthesize suitable fixing actions (Section 4.3.3).

In previous work, we developed Pachika [15], a tool that automatically builds finite-state behavioral models from a set of passing and failing test cases of a Java program. Pachika also generates fix candidates by modifying the model of failing runs in a way which makes it compatible with the model of passing runs. The modifications can insert new transitions or delete existing transitions to change the behavior of the failing model; the changes in the model are then propagated back to the Java implementation. AutoFix exploits some of the techniques used in Pachika—such as finite-state models and state abstraction—in combination with other novel ones—such as snapshots, dynamic analysis for fault localization, fix actions and schema, contracts, and automatic test-case generation.

Weimer [44] presents an algorithm to produce patches of Java programs according to finite-state specifications of a class. The main differences with respect to AutoFix are the need for user-provided finite-state machine specifications, and the focus on security policies: patches may harm other functionalities of the program and "are not intended to be applied automatically" [44].

6.2 Domain-Specific Models

Automated debugging can be more tractable over restricted models of computations. A number of works deal with fixing finite-state programs, and normally assumes a specification given in some form of temporal logic [48], [49].

Gorla et al. [50], [51] show how to patch web applications at runtime by exploiting the redundancy of services offered through their APIs; the patches are generated from a set of rewrite rules that record the relations between services. In more recent work [52], they support workarounds of general-purpose Java applications based on a repertoire of syntactically different library calls that achieve the same semantics.

Janjua and Mycroft [53] target atomicity violation errors in concurrent programs, which they fix by introducing synchronization statements automatically. More recently, Jin et al. [54] developed the tool AFix that targets the same type of concurrency errors.

Abraham and Erwig [55] develop automated correction techniques for spreadsheets, whose users may introduce erroneous formulae. Their technique is based on annotating cells with simple information about their "expected value"; whenever the computed value of a cell contradicts its expected value, the system suggests changes to the cell formula that would restore its value to within the expected range. The method can be combined with automated testing techniques to reduce the need for manual annotations [56].

Samimi et al. [57] show an approach to correct errors in print statements that output string literals in PHP

applications. Given a test suite and using an HTML validator as oracle for acceptable output, executing each test and validating its output induces a partial constraint on the string literals. Whenever the combination of all generated constraints has a solution, it can be used to modify the string literals in the print statements to avoid generating incorrect output. Constraint satisfaction can be quite effective when applied to restricted domains such as PHP strings; along the same lines, AutoFix uses constraint-based techniques when dealing with linear combinations of integer variables (Section 4.3.4).

6.3 Dynamic Patching

Some fixing techniques work *dynamically*, that is at runtime, with the goal of contrasting the adverse effects of some malfunctioning functionality and prolonging the up time of some piece of deployed software. Demsky et al. [58], [59] provide generic support for dynamic patching inside the Java language.

Data-structure repair. Demsky and Rinard [60] show how to dynamically repair data structures that violate their consistency constraints. The programmer specifies the constraints, which are monitored at runtime, in a domain language based on sets and relations. The system reacts to violations of the constraints by running repair actions that try to restore the data structure in a consistent state.

Elkarablieh and Khurshid [61] develop the Juzi tool for Java programs. A user-defined `repOk` Boolean query checks whether the data structure is in a coherent state. Juzi monitors `repOk` at runtime and performs some repair action whenever the state is corrupted. The repair actions are determined by symbolic execution and by a systematic search through the object space. In follow-up work [62], [63], the same authors outline how the dynamic fixes generated by Juzi can be abstracted and propagated back to the source code.

Samimi et al.'s work [64] leverages specifications in the form of contracts to dynamically repair data structures and other applications. As in our work, an operation whose output violates its postcondition signals a fault. When this occurs, their Plan B technique uses constraint solving to generate a different output for the same operation that satisfies the postcondition and is consistent with the rest of the program state; in other words, they *execute the specification* as a replacement for executing a faulty implementation. Their prototype implementation for Java has been evaluated on a few data-structure faults similar to those targeted by Demsky and Rinard [60], as well as on other operations that are naturally expressed as constraint satisfaction problems.

Memory-error repair. The ClearView framework [65] dynamically corrects buffer overflows and illegal control flow transfers in binaries. It exploits a variant of Daikon [66] to extract invariants in normal executions. When the inferred invariants are violated, the system tries to restore them by looking at the differences between the current state and the invariant state. ClearView can prevent the damaging effects of malicious code injections.

Exterminator [67], [68] is a framework to detect and correct buffer overflow and dangling pointer errors in C and C++ programs. The tool executes programs using a probabilistic memory allocator that assigns a memory area of variably larger size to each usage; an array of size n, for example, will be stored in an area with strictly more than n cells. With this padded memory, dereferencing pointers outside the intended frame (as in an off-by-one overflow access) will not crash the program. Exterminator records all such harmless accesses outside the intended memory frame and abstracts them to produce patches that permanently change the memory layout; the patched layout accommodates the actual behavior of the program in a safe way.

7 Conclusions

In the past decade, automated debugging has made spectacular advances: first, we have seen methods to isolate failure causes automatically; then, methods that highlight likely failure locations. Recently, the slogan "automated debugging" has denoted techniques that truly deserve this name: we can actually generate workable fixes completely automatically.

The AutoFix approach, described in the paper, is an important contribution towards the ideal of automatic debugging. In experiments with over 200 faults in software of various quality, AutoFix generated fixes for 42 percent of the faults; inspection reveals 59 percent of them are not mere patches but real corrections of quality comparable to those programmers familiar with the faulty programs could write. AutoFix achieves these results with limited computational resources: running on standard hardware, it required an average time per fix under 20 minutes—where the average includes all failed fixing attempts and the automatic generation of test cases that profile the faults. One of the key ingredients used to achieve these encouraging results is the reliance on *contracts* to boost and automate all debugging steps. The kinds of contracts required by AutoFix are simple and normally available in Eiffel programs; the effort of writing them is, therefore, limited and comparable to other everyday programming activities.

With AutoFix, the programmer's debugging effort could be reduced to almost zero in many cases. We write "*almost* zero", as we still assume that a human should assess the generated fixes and keep authority over the code. One may also think of systems that generate and apply fixes automatically; the risk of undesired behavior may still be preferred to no behavior at all, and can be alleviated by more precise specifications expressed as contracts. In any case, we look forward to a future in which much of the debugging is taken over by automated tools, reducing risks in development and relieving programmers from a significant burden.

Availability

The AutoFix source code, and all data and results cited in this article, are available at: http://se.inf.ethz.ch/research/autofix/

Acknowledgments

This work was partially funded by the Hasler-Stiftung (Grant no. 2327) and by the Deutsche Forschungsgemeinschaft (Ze509/4-1) under the title "AutoFix—Programs that fix themselves"; and by the Swiss National Science

Foundation (Project 200021-134976: "Automated Support for Invariant Inference"). We also gratefully acknowledge the support of the Swiss National Supercomputing Centre (CSCS) for the experiments (Project s264). The concept of generating fixes from differences in passing and failing runs was conceived with Andreas Leitner. Stefan Buchholz and Lucas S. Silva contributed to an early implementation of AutoFix.

REFERENCES

[1] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C.-H. Gros, A. Kamsky, S. McPeak, and D. R. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[2] P. Godefroid, M. Y. Levin, and D. A. Molnar, "SAGE: Whitebox fuzzing for security testing," *Commun. ACM*, vol. 55, no. 3, pp. 40–44, 2012.

[3] J. Penix, "Large-scale test automation in the cloud," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, p. 1122.

[4] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf, "Programs that test themselves," *Comput.*, vol. 42, no. 9, pp. 46–55, Sep. 2009.

[5] EVE: The Eiffel verification environment. (2013). [Online]. Available: http://se.inf.ethz.ch/research/eve/

[6] N. Polikarpova, I. Ciupa, and B. Meyer, "A comparative study of programmer-written and automatically inferred contracts," in *Proc. 18th ACM Int. Symp. Softw. Testing Anal.*, 2009, pp. 93–104.

[7] H.-C. Estler, C. A. Furia, M. Nordio, M. Piccioni, and B. Meyer, "Contracts in practice.," in *Proc. 19th Int. Symp. Formal Methods*, May 2014. [Online]. pp. 230–246. Available: http://arxiv.org/abs/1211.4775

[8] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *Proc. 19th ACM Int. Symp. Softw. Testing Anal.*, 2010, pp. 61–72.

[9] Y. Pei, Y. Wei, C. A. Furia, M. Nordio, and B. Meyer, "Code-based automated program fixing," in *Proc. 26th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2011, pp. 392–395.

[10] N. Polikarpova, C. A. Furia, and B. Meyer, "Specifying reusable components," in *Proc. 3rd Int. Conf. Verified Softw.: Theories, Tools, Experiments*, vol. 6217, Aug. 2010, pp. 127–141.

[11] N. Polikarpova. (2012) EiffelBase2[Online]. Available: http://dev.eiffel.com/EiffelBase2

[12] N. Polikarpova, C. A. Furia, Y. Pei, Y. Wei, and B. Meyer, "What good are strong specifications?" in *Proc. 35rd ACM Int. Conf. Softw. Eng.*, May 2013, pp. 257–266.

[13] B. Meyer, *Object-Oriented Software Construction*. 2nd ed. Englewood Cliffs, NJ, USA: Prentice Hall, 2000.

[14] L. L. Liu, B. Meyer, and B. Schoeller, "Using contracts and Boolean queries to improve the quality of automatic test generation," in *Proc. 1st Int. Conf. Tests Proofs*, 2007, pp. 114–130.

[15] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2009, pp. 550–554.

[16] L. De Moura and N. Bjrner, "Z3: An efficient SMT solver," in *Proc. 14th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2008, pp. 337–340.

[17] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proc. 21st ACM Int. Conf. Softw. Eng.*, 1999, pp. 213–224.

[18] B. R. Liblit, "Cooperative bug isolation," Ph.D. dissertation, Univ. of California, Berkeley, CA, USA:, Dec. 2004.

[19] F. E. Allen, "Control flow analysis," in *Proc. ACM Symp. Compiler Optimization*, 1970, pp. 1–19.

[20] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Mateo, CA, USA: Morgan Kaufmann, 1997.

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

[22] Y. Qi, X. Mao, Y. Lei, and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," in *Proc. ACM Int. Symp. Softw. Testing Anal.*, 2013, pp. 191–201.

[23] Y. L. Chou, *Statistical Analysis*. New York, NY, USA: Holt, Rinehart and Winston, 1975.

[24] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proc. 22nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2007, pp. 433–436.

[25] M. Martinez and M. Monperrus, "Mining software repair models for reasoning on the search space of automated program fixing," *Empirical Software Eng.*, http://link.springer.com/article/10.1007/s10664-013-9282-8, 2013.

[26] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software Engineering," in *Proc. 33rd ACM Int. Conf. Softw. Eng.*, 2011, pp. 1–10.

[27] M. Nordio, C. Ghezzi, B. Meyer, E. D. Nitto, G. Tamburrelli, J. Tschannen, N. Aguirre, and V. Kulkarni, "Teaching software engineering using globally distributed projects: The DOSE course," in *Proc. Collaborative Teaching Globally Distrib. Softw. Develop.—Community Building Workshop*, 2011, pp. 36–40.

[28] Y. Pei, C. A. Furia, M. Nordio, and B. Meyer. (2014, Apr.) "Automatic program repair by fixing contracts," in *Proc. 17th Int. Conf. Fundamental Approaches Softw. Eng.* [Online]. pp. 246–260. Available: http://se.inf.ethz.ch/research/specifix/

[29] K. Pan, S. Kim, and E. J. Whitehead, Jr. (2009, Jun.). Toward an understanding of bug fix patterns. *Empirical Softw. Eng.* [Online]. *14(3)*, pp. 286–315. Available: http://dx.doi.org/10.1007/s10664-008-9077-5

[30] M. Müller, R. Typke, and O. Hagner, "Two controlled experiments concerning the usefulness of assertions as a means for programming," in *Proc. Int. Conf. Softw. Maintenance*, Oct. 2002, pp. 4–92.

[31] K. Beck, *Test-Driven Development*. Reading, MA, USA: Addison-Wesley, 2002.

[32] N. Nagappan, E. M. Maximilien, T. Bhat, and L. Williams, "Realizing quality improvement through test driven development: Results and experiences of four industrial teams," *Empirical Softw. Eng.*, vol. 13, pp. 289–302, 2008.

[33] E. M. Maximilien and L. Williams, "Assessing test-driven development at IBM," in *Proc. 25th Int. Conf. Softw. Eng.*, 2003, pp. 564–569.

[34] 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 *Proc. IEEE 34th Int. Conf. Softw. Eng.*, 2012, pp. 3–13.

[35] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 802–811.

[36] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta, "BugFix: A learning-based tool to assist developers in fixing bugs," in *Proc. 17th IEEE Int. Conf. Program Comprehension*, 2009, pp. 70–79.

[37] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Proc. IEEE Congress Evol. Comput.*, 2008, pp. 162–168.

[38] A. Arcuri, "Evolutionary repair of faulty software," *Applied Soft Comput.*, vol. 11, no. 4, pp. 3494–3514, 2011.

[39] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, 2009, pp. 364–374.

[40] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Commun. ACM*, vol. 53, no. 5, pp. 109–116, 2010.

[41] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan. 2012.

[42] H. He and N. Gupta, "Automated debugging using path-based weakest preconditions," in *Proc. 7th Int. Conf. Fundamental Approaches Softw Eng.*, 2004, pp. 267–280.

[43] D. Gopinath, M. Z. Malik, and S. Khurshid, "Specification-based program repair using SAT," in *Proc. 17th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2011, pp. 173–188.

[44] W. Weimer, "Patches as better bug reports," in *Proc. 5th ACM Int. Conf. Generative Programm. Component Eng.*, 2006, pp. 181–190.

[45] C. Le Goues, W. Weimer, and S. Forrest, "Representations and operators for improving evolutionary software repair," in *Proc. ACM Genetic Evol. Comput. Conf.*, 2012, pp. 959–966.

[46] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 772–781.

[47] S. Chandra, E. Torlak, S. Barman, and R. Bodik, "Angelic debugging," in *Proc. 33rd ACM Int. Conf. Softw. Eng*, 2011, pp. 121–130.

[48] W. Mayer and M. Stumptner, "Evaluating models for model-based debugging," in *Proc. 23rd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2008, pp. 128–137.
[49] B. Jobstmann, S. Staber, A. Griesmayer, and R. Bloem, "Finding and fixing faults," *J. Comput. Syst. Sci.*, vol. 78, no. 2, pp. 441–460, 2012.
[50] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè, "Automatic workarounds for web applications," in *Proc. 18th ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2010, pp. 237–246.
[51] A. Gorla, M. Pezzè, J. Wuttke, L. Mariani, and F. Pastore, "Achieving cost-effective software reliability through self-healing," *Comput. Inform.*, vol. 29, no. 1, pp. 93–115, 2010.
[52] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè, "Automatic recovery from runtime failures," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 782–791.
[53] M. U. Janjua and A. Mycroft, "Automatic corrections to safety violations in programs," in *Proc. Thread Verification Workshop*, 2006, pp. 111–116.
[54] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *Proc. 32nd ACM SIGPLAN Conf. Programm. Language Des. Implementation*, 2011, pp. 389–400.
[55] R. Abraham and M. Erwig, "Goal-directed debugging of spreadsheets," in *Proc. IEEE Symp. Visual Languages Human-Centric Compu.*, 2005, pp. 37–44.
[56] R. Abraham and M. Erwig, "Test-driven goal-directed debugging in spreadsheets," in *Proc. IEEE Symp. Visual Languages Human-Centric Comput.*, 2008, pp. 131–138.
[57] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren, "Automated repair of HTML generation errors in PHP applications using string constraint solving," in *Proc. Int. Conf. Softw. Eng.*, 2012, pp. 277–287.
[58] B. Demsky and A. Dash, "Bristlecone: A language for robust software systems," in *Proc. 22nd Eur. Conf. Object-Oriented Programm.*, 2008, pp. 490–515.
[59] B. Demsky and S. Sundaramurthy, "Bristlecone: Language support for robust software applications," *IEEE Trans. Softw. Eng.*, vol. 37, no. 1, pp. 4–23, Jan. 2011.
[60] B. Demsky and M. Rinard, "Automatic detection and repair of errors in data structures," *ACM SIGPLAN Notices*, vol. 38, no. 11, pp. 78–95, 2003.
[61] B. Elkarablieh and S. Khurshid, "Juzi: A tool for repairing complex data structures," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 855–858.
[62] M. Z. Malik and K. Ghori, "A case for automated debugging using data structure repair," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2009, pp. 620–624.
[63] M. Z. Malik, J. H. Siddiqui, and S. Khurshid, "Constraint-based program debugging using data structure repair," in *Proc. IEEE 4th Int. Conf. Softw. Testing, Verification Validation.*, 2011, pp. 190–199.
[64] H. Samimi, E. D. Aung, and T. D. Millstein, "Falling back on executable specifications," in *Proc. 24th Eur. Conf. Object-Oriented Programm.*, 2010, pp. 552–576.
[65] 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 *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Principles*, 2009, pp. 87–102.
[66] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 99–123, Feb. 2001.
[67] E. D. Berger, "Software needs seatbelts and airbags," *Commun. ACM*, vol. 55, no. 9, pp. 48–53, 2012.
[68] G. Novark, E. D. Berger, and B. G. Zorn, "Exterminator: Automatically correcting memory errors with high probability," *Commun. ACM*, vol. 51, no. 12, pp. 87–95, 2008.

Yu Pei received the PhD degree in computer science from the Department of Computer Science and Technology, Nanjing University, China. He is a research assistant as well as PhD student in the Chair of Software Engineering at ETH Zurich. He was an assistant professor in the Macao University of Science and Technology before he joined the chair. His current research interest includes automated software testing and debugging.

Carlo A. Furia received the Laurea degree in computer science engineering from Politecnico di Milano, the master's of science degree in computer science from the University of Illinois at Chicago, and the PhD degree in computer science from Politecnico di Milano. He is a senior researcher at the Chair of Software Engineering of ETH Zurich. His main research interests include formal methods for software engineering.

Martin Nordio received the bachelor's degree at the University of Rio Cuarto, Argentina and a master's degree at the University of Republica, Uruguay. He joined ETH in July 2005. He is a senior researcher at the Chair of Software Engineering at ETH Zurich. He completed his DrSc. ETH in October 2009 working on proofs and proof transformations for object-oriented programs.

Yi Wei received the MS degree in engineering from Wuhan University in China. He is a post-doc researcher in the Constraint Reasoning group at Microsoft Research Cambridge, United Kingdom. His research interests include dynamic and static program analysis, software testing, and machine learning. Previously, he was an engineer at Eiffel Software and a research assistant at the Chair of Software Engineering at ETH Zurich.

Bertrand Meyer is professor of software engineering at ETH Zurich, head of the Software Engineering Laboratory at ITMO (St. Petersburg, Russia), and a chief architect at Eiffel Software, California.

Andreas Zeller is a full professor for software engineering at Saarland University in Saarbrücken, Germany. His research concerns the analysis of large software systems and their development process. He has received an ERC Advanced Grant, Europe's highest and most prestigious individual research grant, for work on specification mining and test case generation. He is a fellow of the ACM and a member of the IEEE.