Automated Fixing of Programs with Contracts

Yi Wei* · Yu Pei* · Carlo A. Furia* · Lucas S. Silva* · Stefan Buchholz*
Bertrand Meyer* · Andreas Zeller†

*Chair of Software Engineering, ETH Zürich
Zürich, Switzerland
{yi.wei, yu.pei, caf, Bertrand.Meyer}@inf.ethz.ch
{slucas, bustefan}@student.ethz.ch

†Software Engineering Chair
Saarland University – Computer Science
Saarbrücken, Germany
zeller@cs.uni-saarland.de

ABSTRACT

In program debugging, finding a failing run is only the first step; what about correcting the fault? Can we automate the second task as well as the first? The AutoFix-E tool automatically generates and validates fixes for software faults. The key insights behind AutoFix-E are to rely on *contracts* present in the software to ensure that the proposed fixes are semantically sound, and on *state diagrams* using an abstract notion of state based on the boolean queries of a class.

Out of 42 faults found by an automatic testing tool in two widely used Eiffel libraries, AutoFix-E proposes successful fixes for 16 faults. Submitting some of these faults to experts shows that several of the proposed fixes are identical or close to fixes proposed by humans.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids, diagnostics*; D.2.4 [**Software Engineering**]: Software Verification—*programming by contract, assertion checkers, reliability*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program modification*

General Terms

Reliability, Verification

Keywords

Automated debugging, automatic fixing, program synthesis, dynamic invariants

1. INTRODUCTION

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

Automatic testing tools are becoming available to address the first goal. Recently, some progress has also been made towards tools that can propose *automatic corrections.* Such is in particular the goal of the AutoFix project, of which the tool described here, AutoFix-E, is one of the first results. The general idea is, once a testing process (automatic, manual, or some combination) has produced one or more execution failures, reflecting a fault in the software, to propose corrections ("fixes") for the fault. The important question of how to use such corrections (as suggestions to the programmer, or, more boldly, for automatic patching) will not be addressed here; we focus on the technology for *finding good potential corrections.*

The work reported here is part of a joint project, AutoFix, between ETH Zürich and Saarland University. The tool is called AutoFix-E, to reflect that it has been developed for Eiffel, although the concepts are applicable to any other language natively equipped with contracts (such as Spec#) or a contract extension of an existing language (such as JML for Java). The AutoFix project builds on the previous work on automatic testing at ETH, leading to the AutoTest tool [20], and on work on automatic debugging at Saarland, leading in particular to the Pachika tool [5].

While AutoFix-E is only an initial step towards automatic fault correction, the first results, detailed in Section 4, are encouraging. In one experiment, we ran AutoFix-E on 42 errors found on two important libraries (EiffelBase and Gobo) by the AutoTest automatic testing framework [20]. These libraries are production software, available for many years and used widely by Eiffel applications, including ones with up to 2 million lines of code. The use of an automatic testing tool (rather than a manual selection of faults found by human testers) is a protection against bias. AutoFix-E succeeded in proposing valid corrections for 16 of the faults. We then showed some failed executions to a few highly experienced programmers and asked them to propose their own corrections; in several cases the results were the same.

The remainder of this paper is organized as follows: Section 2 illustrates AutoFix-E from a user's perspective on an example program. Section 3 describes the individual steps of our approach. Section 4 evaluates the effectiveness of the proposed fixes. After discussing related work in Section 5, Section 6 presents future work. All the elements necessary to examine and reproduce these and other results described in this article are available for download, as discussed in Section 7: the source code of AutoFix-E, detailed experimental results, and instructions to rerun the experiments.

Listing 1: Calling *duplicate* when *before* holds violates the precondition of *item*.

```
1  duplicate (n: INTEGER): like Current
2          -- Copy of subset beginning at cursor position
3          -- and having at most 'n' elements.
4      require
5          non_negative: n ≥ 0
6      local
7          pos: CURSOR
8          counter: INTEGER
9      do
10         pos := cursor
11         create Result.make
12         from until (counter = n) or else after loop
13             Result.put_left (item)
14             forth
15             counter := counter + 1
16         end
17         go_to (pos)
18     end
19
20 item: G    -- Current item
21     require
22         not_off: (not before) and (not after)
```

2. AN AUTOFIX EXAMPLE

Let us first demonstrate AutoFix-E from a user's perspective. Listing 1 shows an excerpt of the EiffelBase class *TWO_WAY_SORTED_SET*—a collection of objects, with a *cursor* iterating over the collection. The cursor may point before the first or after the last element; these conditions can be queried as boolean functions *before* and *after*.

The *duplicate* routine takes a nonnegative argument n and returns a new set containing at most n elements starting from the current position of *cursor*. After saving the current position of *cursor* as *pos* and initializing an empty set as **Result**, *duplicate* enters a loop that navigates the set data structure and iteratively adds the current *item* to **Result**, calls *forth* to move to the next element in the set, and updates the *counter*.

During automated testing, AutoTest discovered a defect in *duplicate*. The precondition of routine *item* (also shown in Listing 1) requires that **not** *before* and **not** *after* both hold—that is, the cursor is neither before the first element nor after the last. The loop exit condition, however, only enforces the second condition. So if a call to *duplicate* occurs when *before* holds, the call to *item* will violate *item*'s precondition, resulting in a failure.

For this failure, AutoFix-E automatically generates the fix shown in Listing 2, replacing the lines 13–15 in Listing 1. The fix properly checks for the previously failing condition (*before*) and, by calling *forth*, brings the program into a state such that the problem does not occur.

How does AutoFix-E generate this fix? Two properties in particular distinguish AutoFix-E from previous work in the area, namely the reliance on *contracts* and on *boolean query abstraction*:

Contracts associate specifications, usually partial, with software elements such as classes or routines (methods). The work on AutoTest has already shown that contracts significantly boost automatic testing [4].

Listing 2: Fixed loop body generated by AutoFix-E

```
13         if before then
14             forth
15         else
16             Result.put_left (item)
17             forth
18             counter := counter + 1
19         end
```

AutoFix-E illustrates a similar benefit for automatic correction: as a failure is defined by a contract violation, knowledge about the contract helps generate a fix not just by trying out various syntactical possibilities, but by taking advantage of deep semantic knowledge about the intent of the faulty code. For example, if the evidence for the failure is that an assertion q is violated, and analysis determines that a certain routine r has q as part of its postcondition, adding a call to r may be a valid fix.

Boolean queries of a class (boolean-valued functions without arguments) yield a partition of the corresponding object state into a much smaller set of abstract states. This partition has already proved useful for testing [15], and it also helps for automatic correction, by providing the basis for state/transition models.

These two properties form the core of our approach, detailed in the following section.

3. HOW AUTOFIX WORKS

Let us now describe the detailed steps of our approach. Figure 1 summarizes the individual steps from failure to fix, detailed in the following subsections.

3.1 Contracts and Correctness

AutoFix-E works on Eiffel classes [8] equipped with *contracts* (made of *assertions* [19]). Contracts constitute the specification of a class and consist of preconditions (**require**), postconditions (**ensure**), intermediate assertions (**check**), and class invariants (translated into additional postcondition and precondition clauses in all the examples).

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 that routine's precondition.

3.2 Testing and Fault Discovery

Testing a routine with various inputs can reveal a fault in the form of an assertion violation. Testing can be manual—with the programmer providing a number of test cases exercising a routine in ways that may make it fail—or automatic. In previous work, we developed AutoTest [20], a fully automated random testing framework for Eiffel classes. AutoTest generates as many calls as possible in the available testing time, and reports faults in the form of minimal-length failing runs. In practice, AutoTest regularly finds faults in production software [4]. The faults found by AutoTest are a natural input to AutoFix-E; the integration of

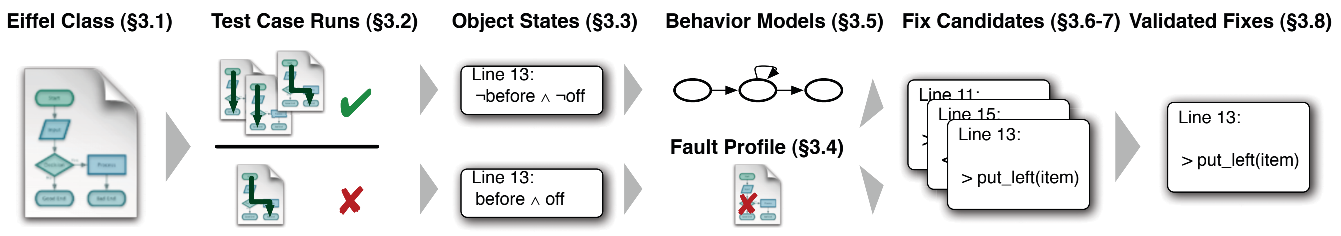

Figure 1: How AutoFix-E works. We take an Eiffel class (Section 3.1) and generate test cases with AutoTest (Section 3.2). From the runs, we extract object states using boolean queries (Section 3.3). By comparing the states of passing and failing runs, we generate a *fault profile* (Section 3.4), an indication of what went wrong in terms of *abstract object state.* From the state transitions in passing runs, we generate a *finite-state behavioral model* (Section 3.5), capturing the normal behavior in terms of *control.* Both control and state guide the generation of *fix candidates* (Section 3.6), with special treatment of linear assertions (Section 3.7). Only those fixes passing the regression test suite remain (Section 3.8).

the two tools is intended to provide a completely automated "push-button" solution to the testing-debugging phases of software development. AutoFix-E can, however, work with any set of test cases exposing a fault, whether these cases have been derived automatically or manually.

3.3 Assessing Object State

To generate fixes, we first must reason about where a failure came from. For this purpose, we assess object state by means of boolean queries.

3.3.1 Argument-less Boolean Queries

A class is usually equipped with a set of argument-less, 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 an item or "off" all items, a checking account is overdraft or not. n boolean queries for a class define a partition of the corresponding object state space into 2^n "abstract states". As n is rarely higher than about 15 in practice,[1], the resulting size is not too large, unlike the concrete state space which is typically intractable. Intuitively, such a partition is attractive, because for well-designed classes, boolean queries characterize fundamental properties of the objects. Our earlier work [15] confirms this intuition for testing applications, showing that high boolean query coverage correlates with high testing effectiveness; our earlier work on fix generation [5] also relied on similar abstractions. The reasons why state abstraction is effective appear to be the following:

- Being argument-less, boolean queries describe the object state *absolutely*, as opposed to in relation with some given arguments.
- Boolean queries usually do not have any precondition, hence they posses a definite value in any state. This feature is crucial to have a detailed object state at any point in a run.
- Boolean queries are widely used in Eiffel contracts, which suggests that they model important properties of a class.

[1]In EiffelBase [9], 284 out of 305 classes (94%) have 15 or fewer boolean queries.

In all, it is very likely that there exists some boolean query whose value in failing runs is different than in error-free runs.

We have seen some examples of useful boolean queries in Section 2: *before* and *after* describe whether the internal cursor is before the first element or after the last element in an ordered set. *is_empty* is another boolean query of the class which, as the name indicates, is true whenever the set contains no elements.

3.3.2 Complex Predicates

The values of boolean queries are often correlated. Our experience suggests that *implications* can express most correlations between boolean queries. For example, executing the routine *forth* on an empty set changes *after* to true; in other words, the implication

$$\textit{is\_empty}\ \textbf{implies}\ \textit{after}$$

characterizes the effect of routine *forth*. Such implications are included as predicates for the object state model, because they are likely to make the abstraction more accurate.

Trying out all possible implications between pairs of boolean queries is impractical and soon leads to a huge number of often irrelevant predicates. We therefore use two sources for implications:

Contracts. We mine the *contracts* of the class under analysis for implications; these implications are likely to capture connections between boolean queries.

Mutations. We also found it useful to generate three *mutations* for each implication, obtained by negating the antecedent, the consequent, or both. These mutations are often helpful in capturing the object state in faulty runs.

In our ongoing example, the implication *is_empty* **implies** *after* mutates into: (1) **not** *is_empty* **implies** *after*; (2) *is_empty* **implies not** *after*; and (3) **not** *is_empty* **implies not** *after*.

The resulting set of predicates, including boolean queries and implications, is called *predicate set* and denoted by P; Π denotes the set $P \cup \{\textbf{not}\ p \mid p \in P\}$ including all predicates in the predicate set and their negations.

3.3.3 Predicate Pruning

The collection P of boolean queries and implications usually contains *redundancies* in the form of predicates that are co-implied (i.e., they are always both true or both false) or contradictory (i.e., if one of them is true the other is false and vice versa). Some predicates may also be *subsumed* by class invariants. Redundant predicates increase the size of the predicate set without providing additional information.

To prune redundant predicates out of the predicate set P, an automated theorem prover is the tool of choice. We use Z3 [6] to check whether a collection of predicates is contradictory or valid, and remove iteratively redundant predicates until we obtain a suitable set—neither contradictory nor valid. While there is in general no guarantee that this procedure gives a minimal set, the relationships among predicates are sufficiently simple that this unsophisticated approach gives good results.

3.4 Fault Analysis

Once we have extracted the predicates, we need to characterize the *crucial difference* between the passing and failing runs. To characterize this difference, AutoFix-E runs Daikon [11] to find out which of the predicates in Π (determined in Section 3.3) hold over all passing and failing runs, respectively.[2] These *state invariants* characterize the *passing state* and *failing state*, respectively; their difference outlines the failure cause.

3.4.1 Fault Profiles

We first consider all passing runs and infer a state invariant at each program location that is executed. An invariant is a collection of predicates from Π, all of which hold at the given location in every passing run. Then, we perform the same task with the set of failing runs for the bug currently under analysis; the failing runs cannot continue past the location of failure, hence the sequence of invariants will also stop at that location. The results of this phase are *two sequences of invariants*, a pair for each executed location up to the point of failure.

Comparing these pairs of failing and passing states results in a *fault profile*. The fault profile contains all predicates that hold in the passing run, but not in the failing run. This is an an alleged indication of what "went wrong" in the failing run in terms of abstract object *state*.

In the case of routine *duplicate*, AutoFix-E finds out that, before the location of the exception, the state invariant *before* **and** *off* holds only in failing test cases. Correspondingly, the fault profile is the complement predicate:

before **implies** **not** *off*

Either of the predicates *before* or *off* may be the cause of the the precondition violation. To narrow down the cause further, we use two heuristics:

1. We consider only predicates in the failing state invariants that imply the negation of the violated assertion.

2. Among those remaining predicates, we attempt to find out the strongest (if possible).

[2]While Daikon also reports other properties as invariants, we are only interested in the boolean query predicates as discussed in Section 3.3.

Again, Z3 does the reasoning. In our example, both predicates imply the negation of the violated precondition, so the first heuristic does not help. The second heuristic, though, finds out that *before* **implies** *off* in failing runs. Being the antecedent of the implication, *before* is very likely to be the cause of the assertion violation.

Formally, the fault profile is constructed as follows: For a given location ℓ, let $I_\ell^+ \subseteq \Pi$ and $I_\ell^- \subseteq \Pi$ be the sets of predicates characterizing the passing and failing state, respectively. The *fault profile* at location ℓ is the set $\Phi_\ell \subseteq \Pi$ of all predicates $p \in \Pi$ for which $p \in I_\ell^+ \wedge p \notin I_\ell^-$ holds; it characterizes potential causes of errors at ℓ.

3.4.2 Fixing as a Program Synthesis Problem

Using the fault profile at the program point where a failure occurs, we can formulate the problem of finding a fix as a *program synthesis* problem. For a location ℓ, let *fail_state* and *pass_state* be the invariants characterizing the failing and passing runs, respectively. Any piece of code that can drive the program state from *fail_state* to *pass_state* is a candidate fix which, if executed at ℓ, would permit normal continuation of the failing run. Formally, we are looking for a program *fix* that satisfies the specification:

require *fail_state* **do** *fix* **ensure** *pass_state*

The underlying assumption is that program *fix* is in general sufficiently simple—consisting of just a few lines of code—and that, consequently, it can be generated automatically.

3.5 Behavioral Models

Now that we can characterize the difference between passing and failing state, we need to know how to *reach* the passing state—that is, how to synthesize a fix. Building on our previous experiences with Pachika [5], AutoFix-E extracts a simple *finite-state behavioral model* from *all* error-free runs of the class under analysis.

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. Transitions are labeled with routine names: a routine m with specification **require** *pre* **ensure** *post* appears on transitions from a state where *pre* holds to a state where *post* holds.

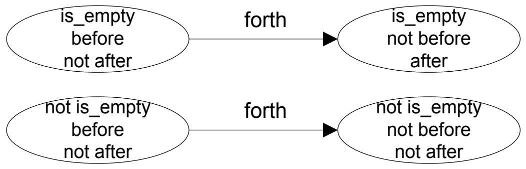

Figure 2: State transitions of *forth*

As an example, Figure 2 shows the behavioral model for the *forth* routine from our running example. We can see that if the initial state was *is_empty*, *after* will always hold after a call to *forth*. Also, after invoking *forth*, **not** *before* will always hold. Therefore, if we want to reach the **not** *before* state, as we would do in our running example to avoid violating the precondition, invoking *forth* is a possible option.

In general, the built abstraction is neither complete nor sound because it is based on a finite number of test runs.

Nonetheless, our experiments showed that it is often sufficiently precise to guide the generation of valid fixes.

We call *mutator* any routine that changes the state according to the behavioral model; a sequence of mutators (reaching a specific state) is called a *snippet*.

3.5.1 Reaching States

To derive a possible fix from a behavioral model, we need to determine sequences of routine calls which can change the object state appropriately. Formally, this works as follows.

For every predicate $p \in \Phi_\ell$ in a fault profile, M^p denotes the set of *mutators*: the routines that, according to the finite-state model, drive the object from a state where p does not hold to a state satisfying p. For a set of predicates ϕ, M^ϕ denotes the set of common mutators $\bigcap_{p\in\phi} M^p$.

A *snippet* for a set of predicates $\{p_1, p_2, \ldots\}$ is any sequence of mutators $\langle m_1, m_2, \ldots\rangle$ that drive the object from a state where none of $\{p_1, p_2, \ldots\}$ holds to one where all of them hold. It may be the case that a routine is the mutator for more than one predicate, hence the length of the snippet need not be the same as the size of the set of predicates it affects. For the running example, a predicate *off* is equivalent to *before* **or** *after*, hence any mutator for *before* or *after* is also a mutator for *off*.

$S[\Phi_\ell]$ denotes the set of all snippets for the set or predicates Φ_ℓ:

$$S[\Phi_\ell] \quad \triangleq \quad \bigcup_{\substack{\{\phi_1,\phi_2,\ldots\}\in\mathsf{Part}(\Phi_\ell)\\ m_1\in M^{\phi_1}, m_2\in M^{\phi_2},\ldots}} \langle m_1, m_2, \ldots\rangle$$

where $\mathsf{Part}(V)$ denotes the set of all partitions (of any size) of V.

Since the finite-state abstraction is built from observing individual mutators, not all snippets may actually be executable or achieve the correct result. Invalid snippets will, however, be quickly discarded in the validation phase. We found that being permissive in the snippet generation phase is a better trade-off than adopting more precise—but significantly more expensive—techniques such as symbolic execution or precise predicate abstraction.

3.6 Generating Candidate Fixes

Now that we know about fault-characterizing states (Section 3.4) and mutators between these states (Section 3.5), we can leverage them to actually generate fixes. AutoFix-E builds a set of possible candidate fixes in two phases. First, it selects a *fix schema*—a template that abstracts common instruction patterns. Then, it instantiates the fix schema with *actual conditions*, as extracted from the fault profile (Section 3.4), and routine calls, as obtained from the behavioral model (Section 3.5). The set of possible instantiations constitutes a set of *candidate fixes*.

3.6.1 Fix Schemas

To restrict the search space for code generation, AutoFix-E uses a set of predefined templates called *fix schemas*. The four fix schemas currently supported are shown in Table 1. In these schemas, *fail* is instantiated by a predicate, *snippet* is a sequence of mutators to reach a state, and *old_stmt* some statements, in the original program, related to the point of failure.[3]

[3]The AutoFix-E framework can accommodate different implementations of these two components and, more generally, different techniques for the generation of fix candidates from the specification. Thus, advances in any of these techniques (e.g., [23]) immediately translate into a more powerful automated debugging technique.

(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

Table 1: Fix schemas implemented in AutoFix-E.

(b) **if** *before* **then**
    forth
end
Result.*put_left* (*item*)
forth
counter := *counter* + 1
	(d) **if** *before* **then**
	    forth
	else
	    Result.*put_left* (*item*)
	    forth
	    counter := *counter* + 1
	end

Table 2: Fix candidates. By instantiating the fix schemas (b) and (d) from Table 1 for the *duplicate* example (Listing 1), we obtain two fix candidates.

For the running example, AutoFix-E generates two candidate fixes by instantiating the fix schemas (b) and (d), respectively; they correspond to the routine *duplicate* with the two sequences of statements in Table 2 replacing the original loop body.

3.6.2 Instantiating Schemas

The instantiation of schemas starts from the location where the fault occurred and might possibly backtrack to previous locations in the failing run if no valid fix can be generated at the point of failure.

The instantiation of schemas for a location ℓ is done exhaustively as follows:

fail takes one of the following values:

1. **not** p, for a single predicate $p \in \Phi_\ell$;
2. **not** p_1 **and not** p_2 **and** ..., for a selection of predicates $p_1, p_2, \ldots \in \Phi_\ell$;
3. **not** *violated_clause*, where *violated_clause* is the originally violated assertion clause revealing the fault.

The first case is the most common, and is sufficient in the large majority of cases. The second case does not necessary lead to a huge number of selections, because Φ_ℓ is often a small set. The third case is useful when the predicates in Π cannot precisely characterize the violated assertion that caused the failure; in these cases, we simply replicate verbatim the very clause that was violated.

snippet takes any value in $S[\Phi_\ell]$.

old_stmt takes one of the following values:

1. the lone statement at location ℓ;
2. the block of statements that immediately contains ℓ.

Accordingly, the instantiated schema replaces the statement at location ℓ or the whole block.

3.7 Linearly Constrained Assertions

In contract-based development, many assertions take the form of *linear constraints*—that is, an assertion consisting of boolean combinations of linear inequalities over program variables and constants. As an example of such *linearly constrained assertions* (or *linear assertions* for short), consider the precondition of routine *put_i_th*, which inserts item v at position i in a sorted data structure:

```
put_i_th (v: G; i: INTEGER) require i ≥ 1 and i ≤ count
```

The precondition requires that i denotes a valid position in the range $[1..\ count]$, where *count* is the total number of elements in the structure.

As linear assertions are common in contracts, they require special techniques for fix generation in case they are violated. The rest of this section describes them succinctly and illustrates how they are integrated within the general automated debugging framework.

3.7.1 Fault Analysis

The violation of a linear assertion occurs when a variable takes a value that does not satisfy the constraint. In the example, calling routine *put_i_th* with $i = 0$ triggers a precondition violation.

A fix for a linear assertion violation should check whether the variable subject to the linear constraint takes a value compatible with the constraint itself and, in case it does not happen, it should change the value into a valid one. This means that we cannot treat linear constraints as atomic propositions, but we must "open the box" and reason about their structure. More precisely, it is important to determine what is a *variable* and what is a *constant* in a linear assertion. A variable is the parameter that must be changed (i in the example), while everything else is a constant—with respect to the constraint—and should not be modified (*count* and, obviously, 1 in the example).

If we can tell variables and constants apart, constraint solving can determine a valid value for variables: an extremal solution to the constraint, expressed as a symbolic expression of the constants. Finally, injecting the valid value into the routine using suitable fix schemas produces a candidate fix.

3.7.2 Variable Selection

Some heuristics can guess which identifiers occurring in a linear assertions are variables and which are constants. The heuristics assign a weight to each identifier according to the following guidelines; the identifier with the least weight will be the variable, and everything else will be a constant.

- Routine arguments in preconditions receive lower weights than other identifiers.
- In any assertion, an identifier receives a weight inversely proportional to the number of its occurrences in the assertion (e.g., appearing twice weighs less than appearing once).
- Identifiers that the routine body can assign to receive less weight than expressions that cannot change.

To account for the imperfection of this weighing scheme, it is useful to introduce a bit of slack in the selection of the variable and trying out more than one candidate among the identifiers with low weights. If two different selections both work as a fix, the one with the lowest weight will eventually be chosen as the best fix in the final ranking of valid fixes.

3.7.3 Fixes for Linear Assertions

Fixes for linear assertions are generated in two phases: first, we select a value for the variable that satisfies the constraint, and then we plug the value into a fix schema and inject it in the routine under analysis.

Given a linear assertion λ and a variable v, we look for extremal values of v that satisfy λ. AutoFix-E uses Mathematica [17] to solve λ for maximal and minimal values of v as a function of the other parameters (numeric or symbolic) in λ. To increase the quality of the solution, strengthen λ with linear assertions from the class invariants which share identifiers with λ. In the example of *put_i_th*, the class invariant $count \geq 0$ is added to λ when looking for extrema. The solution consists of the extremal values 1 and *count*.

We use the following schema to build candidate fixes for linear assertions.

```
if not constraint then new_stmt else old_stmt end
```

The violated linear assertion replaces *constraint* in the schema. The rest of the schema is instantiated as follows, for an extremal value *ext* for variable v.

Precondition violation: *old_stmt* is the routine invocation triggering the fault; *new_statement* is a call to the same routine with *ext* replacing v; the instantiated schema replaces the faulty routine invocation in the candidate schema.

Postcondition or other assertion violation: *old_stmt* is empty, *new_statement* is the assignment $v := ext$, and the instantiated schema precedes the **check** statement or the end of the routine.

A faulty call *put_i_th* $(x,\ j)$ is a case of precondition violation, which we would handle by replacing it with:

```
if not ( j ≥ 1 and j ≤ count ) then
        put_i_th (x, 1)
else
        put_i_th (x, j)
end
```

3.8 Validating Fixes

The previous sections have shown how to generate a number of *candidate fixes*. But do these candidates actually correct the fault at hand? For this purpose, AutoFix-E runs all the candidates through the full set of test cases and retains those that pass all runs. A fix is *valid* if it passes all the (previously) failing test cases and it still passes the original passing test cases generated in the fault analysis. Given that the contracts constitute the specification, we can even call the fix a *correction.*

In the example of routine *duplicate*, we generated two candidate fixes, shown in Table 2. These two candidates are now put to the test:

- The left fix (b) still does not pass test cases where *before* holds. In test cases where the set is empty, the fix fails because the first call to *forth* makes *after* true, which causes the violation of the precondition **require not** *after* of routine *forth* at its second call.

  The left fix also fails on non-empty sets because two consecutive calls to *forth* skip one element of the set; this misbehavior will cause a violation of the postcondition of routine *duplicate*.

- The right fix (d) of routine *duplicate* is instead a valid fix: it does not change the behavior of the loop if *before* is false, and it moves the cursor to the first element if *before* is true, so that all references to *item* are error-free.

Consequently, fix (d) is retained, and finally suggested to the programmer as a valid fix (Listing 2).

3.9 Ranking Fixes

AutoFix-E often finds *several* valid fixes for a given fault. While it is ultimately the programmer's responsibility to select which one to deploy, flooding her 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-E ranks the valid fixes according to two simple metrics, combining dynamic and static information.

The experiments in Section 4 will show that these metrics are sufficient to guarantee that, in the large majority of cases, a "proper" fix appears within the first five fixes in the ranking. Here "proper" refers to the expectations of a real programmer who is familiar with the code-base under analysis.

3.9.1 Dynamic Metric

Dynamic (semantic) metrics prefer fixes which modify the run-time behavior of passing test cases as little as possible; the intuition is that a good fix does not affect significantly the behavior of correct runs of the program.

The dynamic metric estimates the difference in runtime behavior between the fix and the original (faulty) program over the set of originally passing runs. It is based on *state distance*, defined as the number of argument-less boolean and integer queries whose value differ in two object states.

AutoFix-E sums the state distances over all routine exit points in all passing runs for the original and the fixed programs; the final figure gives the value of the dynamic metric.

3.9.2 Static Metric

Static (syntactic) metrics favor fixes with simpler textual changes; the smaller its value, the smaller the textual change in the source code. Fixes that introduce only minor textual changes are likely easier to understand and maintain, hence preferable—all else being equal—over more complex ones.

We use a simple static metric which combines three syntactic measures according to the formula:

$$\widehat{\mathrm{OS}} + 5 \cdot \widehat{\mathrm{SN}} + 2.5 \cdot \widehat{\mathrm{BF}}$$

where each weighted factor is the normalized value of the corresponding syntactic measure. The weights have been determined empirically, while the three measures are defined as follows, with reference to the fix schemas in Table 1.

***Old Statements (OS)*:** zero for the fix schemas (a),(b), and the number of statements in *old_stmt* for the fix schemas (c),(d).

This factor measures the number of original instructions that the fix "encloses"; schemas (a) and (b) execute *old_stmt* unconditionally, hence they score zero for this factor.

***Snippet Size (SN)*:** number of statements in *snippet*.

This factor measures the complexity of the new instructions used by the fix.

***Branching Factor (BF)*:** number of branches to reach *old_stmt* from the point of injection of the instantiated fix schema.

This factor measures how deep does the fix "bury" the original instructions within the fix schema.

4. EXPERIMENTAL EVALUATION

This section reports on some experiments that applied AutoFix-E to several faults found in production software and provides a preliminary assessment of the quality of the automatically generated fixes.

4.1 Experimental Setup

All the experiment ran on a Windows 7 machine with a 2.53 GHz Intel dual-core CPU and 4 GB of memory. AutoFix-E was the only computationally-intensive process running during the experiments. On average, AutoFix-E ran for 2.6 minutes for each fault.

4.1.1 Selection of Faults

We ran AutoFix-E on 42 faults detected by AutoTest in 10 data structure classes from the EiffelBase and Gobo libraries [9, 12]. Table 3 lists, for each class, its length in lines of code (LOC), its number of routines (#R), of boolean queries (#B), and the number of faulty routines (#F).

Table 3: Classes used in the experiments.

Class	LOC	#R	#B	#F
ACTIVE_LIST	2,547	155	16	3
ARRAYED_CIRCULAR	1,912	131	15	9
ARRAYED_LIST	2,357	149	16	1
ARRAYED_QUEUE	1,654	109	10	1
ARRAYED_SET	2,705	162	12	2
ARRAY	1,354	93	11	5
BOUNDED_QUEUE	876	63	11	4
DS_ARRAYED_LIST	2,762	166	8	7
LINKED_PRIORITY_QUEUE	2,375	123	10	3
TWO_WAY_SORTED_SET	2,871	139	16	4

The selection covers varied types of faults (Table 4 groups them according to the type of assertion they violate), in routines of diverse complexity (Table 5, where the same routine can originate more than one fault).

Table 4: Types of faults and fixes.

Type of fault	# Faults	# Fixed	# Proper
Precondition	24	11 (46%)	11 (46%)
Postcondition	8	0	0
Check	1	1(100%)	0
Class invariant	9	4 (44%)	2 (22%)
Total	**42**	**16 (38%)**	**13(30%)**

Table 5: Lines Of Code (LOC) of faulty routines.

LOC	1–5	6–10	11–20	21–30	31–40	**Total**
# Rout.	15	11	9	3	1	**39**

4.1.2 *Selection of Test Cases*

The selection of test cases for passing (and failing) runs can affect significantly the performance of automated debugging. To reduce this bias and to show that the whole chain "testing *and* fixing" can be fully automated, our experiments only use test cases generated automatically by AutoTest. AutoFix-E is allowed to discard test cases if they are "redundant" and do not add information to the finite-state abstraction; this filters out several test cases produced by AutoTest without affecting the quality of the generated fixes. In our experiments, the average number of passing and failing test cases for a fault is 9 and 6.5, respectively.

4.2 Results

Valid fixes. The column "# Fixed" in Table 4 shows the number of faults for which AutoFix-E built at least one valid fix. The data shows that AutoFix-E works better on precondition and class invariant violations than on other types of fault. AutoFix-E failed an all of the 8 postcondition violation faults because they all involved complex assertions that could not be characterized precisely enough in terms of boolean queries. While future work will address such limitations, it is interesting to notice that, in our experience with random testing, precondition violations occur significantly more frequently than postcondition violations.

Candidate fixes. The number of candidate fixes generated by AutoFix-E for a given fault varies wildly, ranging from just a couple for linear assertions up to over a thousand for assertion violations occurring within a complex loop. For each of the 16 faults which were fixed automatically, AutoFix-E generated, on average, 165.75 candidates and 12.1 valid fixes; the average percentage of valid fixes per candidate is instead 22%.

Assertion forms. Table 6 correlates the form of a violated assertion—whether it is a single boolean query, an implication, a linear constraint, etc.—and the success of AutoFix-E in finding valid fixes for that form. For each assertion form, the second column reports the number of automatically fixed faults. AutoFix-E is most effective for simpler assertions consisting of a single boolean query or a linear constraint. In both cases, it is straightforward to track down the "causes" of a fault and there is a higher chance of finding routine calls that restore a valid state.

Table 6: Relationship between assertion form and valid fixes.

Assertion form	# Faults	# Fixed
Single boolean query	6	5 (83%)
Implication	2	1 (50%)
Linear	7	5 (71%)
Other[4]	27	5 (19%)

Quality of fixes from a programmer's perspective. A valid fix is only as good as the test suite it passes. As a sanity check on the automatically generated fixes, we manually inspected the top five valid fixes for each fault, according to the ranking criterion of AutoFix-E. The rightmost column of Table 4 counts the number of faults for which we could find at least one "proper" fix among the top five. A "proper" fix is one that fixes the bug without obviously introducing other bugs; it might still not be the best correction, but it is a significant improvement over the buggy code and not just a trivial patch. In all the experiments, whenever a proper fix exists it ranks first among the valid fixes, which gives us some confidence in the ranking criteria implemented in AutoFix-E.

As an additional preliminary assessment of the quality of the automatically generated fixes, we selected a few faults and asked two experienced programmers from Eiffel Software to write their own fixes for the faults. In 4 out of 6 of the cases, the programmers submitted fixes which are identical (or semantically equivalent) to the best fixes produced automatically by AutoFix-E.

As an example of multiple valid fixes for the precondition violation in routine *duplicate*, consider the following two fragments, replacing the loop (lines 12–16 in Listing 1).

```
from                          from
  if before then                if before then
      forth                         start
  end                           end
until ... loop                until ... loop
  -- original loop body         -- original loop body
end                           end
```

AutoFix-E discovered these two valid fixes, but ranked them lower than the one presented in Section 2: the static metric prefers corrections that modify locations closest to the failing statement (inside the **loop** body in the example) over those modifying other statements (the **from** clause). The two new fixes are perfectly equivalent because *forth* and *start* have the very same effect on an object where *before* holds. Semantically, they are also equivalent to the one previously shown, as *forth* is executed only once because it is guarded by *before*. These two fixes, however, achieve a better run-time performance (in the absence of sophisticated compiler optimizations) as they check *before* only once. The expert programmers indeed suggested the rightmost one as a correction. The human preference for using *start* over the equivalent, in this context, *forth* is probably due to the name of the two routines, where *start* suggests a closer relation to the notion of *before* than *forth* does.

[4]Most of the "other" assertions include queries with arguments or implications mixing queries and linear constraints.

4.3 Threats to Validity

The following threats may influence the generalization of our results:

- All the classes used in the experiments are data structure related. Although they have heterogeneous semantic and syntactic complexities, they are not necessarily representative of programs in general. The effectiveness of AutoFix-E may be quite different when applied to other classes.
- The assertions being violated in the examples may also reflect the particular contracting style of these libraries; for example they often refer to the notion of cursor in a data structure. We do not know if this might bias the results.
- The evaluation of proper fixes was done under the assumption that contracts present in classes are correct and strong enough. This may not be true from the point of view of the library developers, who may disagree with some fixes we classified as proper and suggest, instead, changes in the contract as proper fixes in those cases.
- We have not yet performed a large-scale retro-analysis of inferred fixes against fixes actually performed in the history of a project. This is part of planned future work.

5. RELATED WORK

This section reviews techniques that correct programming errors automatically by combining diverse software engineering techniques.

5.1 Restricted Models

Automated debugging is more tractable for restricted models of computations; a number of works deal with fixing finite-state programs automatically (see e.g., [18, 24]).

Abraham and Erwig developed automated debugging techniques for spreadsheets, where the user may introduce erroneous formulas. In [1], they present a technique 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. Their method can be combined with automated testing techniques to reduce the need for manual annotations [2].

5.2 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.

Data structure repair. Demsky and Rinard [7] 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 develop the Juzi tool for Java programs [10]. 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 action is based on symbolic execution and a systematic search through the object space. In a paper presenting ongoing work [16], the same authors outline how the dynamic fixes generated by the Juzi tool could be abstracted and propagated back to the source code.

Memory errors repair. The ClearView framework [22] dynamically corrects buffer overflows and illegal control flow transfers in binaries. It exploits a variant of Daikon to extract invariants in normal executions. When the inferred invariants are violated, the system tries to restore them from a faulty state by looking at the differences between the two states. ClearView can prevent the damaging effects of malicious code injections.

Exterminator [21] is another tool for dynamic patching of memory errors such as out-of-bound. It monitors repeated runs of a program and allocates extra memory to accommodate out-of-bound references appropriately.

5.3 Static Debugging

Static approaches to automated debugging target the source code to permanently remove the buggy behavior from a program.

Model-driven debugging. Some automated debugging methods — including the one in the present paper — rely on the availability of a finite-state abstraction of the program's behavior to detect errors and build patches.

Weimer [25] presents an algorithm to produce patches of Java programs according to finite-state specifications of a class. The main differences with respect to our work 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" [25].

Machine-learning-based debugging. Machine-learning techniques can work in the absence of user-provided annotations, which is a big advantage to analyze pre-existing legacy code in languages such as C or Java.

Jeffrey et al. [14] 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.

Other authors applied genetic algorithms to generate suitable fixes. Arcuri and Yao [3] 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. [26] present a genetic algorithm that takes as input an unannotated program, a set of successful test cases, and a single failing one. After rounds of evolution, the program changes into one that passes all test cases, including the previously failing one. [26] mitigates the limited scalability of genetic algorithms by only producing changes that re-use portions of code available elsewhere in the program and that are limited to the portions of code that is executed when the bug occurs. This, however, requires that the failing execution path is different than the successful execution path, a restriction which does not apply to our approach. Another limitation of [26] resides in its sensitivity to the quality (and size) of the provided test suite, an effect which is much more limited in our approach where random testing techniques can generate a suitable test suite automatically.

Axiomatic reasoning. He and Gupta [13] present a technique that compares two program states at a faulty location in the program. Unlike our work, [13] computes these characterizations statically with weakest precondition reasoning. 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. Besides the apparent high-level similarities between the approach of [13] and ours, there are several important details that differ. First, weakest precondition reasoning requires a very detailed postcondition (e.g., full functional specifications in first-order logic), which also limits scalability. In addition, [13] computes patches by *syntactically* comparing the two program states; this restricts the fixes that can be automatically generated to limited changes in expressions (for example in off-by-one errors).

5.4 Our Earlier Work

In [5], Dallmeier, Zeller, and Meyer presented Pachika, a tool that automatically builds finite-state behavioral models of a set of passing and failing test cases of a Java class. A fix candidate is then a modification of the model of failing runs which makes it compatible with the model of passing runs. In practice, Pachika can insert new transitions or delete existing transitions to change the behavior of the failing model.

While Pachika and AutoFix-E share common origins as part of the joint AutoFix project (such as behavior models, state abstraction, and creating fixes from transitions), AutoFix-E brings several advances, which constitute the contributions of this paper. The main contribution is that AutoFix-E leverages *user-provided contracts* integrating them with dynamically inferred information; this shapes and impacts all stages of fix generation (Section 3). In addition, the concepts of *fix schemas* (Section 3.6), *linear assertions* (Section 3.7), and *ranking fixes* (Section 3.9) are unique to the present paper. All these contributions imply a greater flexibility in accommodating different templates, more complex types of fix actions, and a much higher effectiveness and efficiency in generating fixes.

6. FUTURE WORK

Much remains to be done in automatic fix generation as in our own tool. Our future work will focus on the following topics:

Retro-analysis. For further validation of the approach, we plan to run a systematic application of AutoFix-E on the history of fixed bugs in representative projects, to compare the results with the fixes that developers actually applied.

Contract-less environments. In the absence of contracts, defects are much harder to locate. We want to leverage dynamic invariants and state models to identify likely violations and derive locations for potential fixes.

Predicate selection and instantiation. If one only includes predicates that are relevant to a particular fault in the state model, this can result in fewer, yet more relevant fixes to be generated. Also, the current implementation of AutoFix-E does not instantiate fix schemas with more than one predicate at a time; all these options are on our agenda.

Improved behavior models. Our technique relies on boolean queries to build a state model (Section 3.3), hence it works poorly for classes without boolean queries or such that boolean queries do not capture effectively salient features of the object state.

Dependence on executions. We build a finite-state behavioral model using test runs randomly generated by AutoTest (Section 3.5). This might lead to inconsistent or biased behavioral models, which would then generate ineffective or needlessly complicated snippets. The current implementation reduces the chances of this happening to a minimum with validation (Section 3.9) and by empirically limiting the maximum admitted length of a snippet to a small number (typically two or three instructions). We want to apply more sophisticated techniques to build the behavioral model—maybe with some soundness or minimality guarantee—to improve performances.

Enriching the execution space. As “causes” for a fault are tracked down by comparing abstract object states during passing and failing runs, it is impossible to work on a single fault report without a sufficient number of passing runs. Of course, it is always possible to use automated random testing to build a sufficient number of passing test cases. This limitation does not apply to linear constraints (Section 3.7).

Alternate fault types. We only handle faults in the form of assertion violations. We plan to extend our approach to consider different kinds of faults, such as *Void* dereferencing, integer overflow, I/O errors, etc.

Faults in contracts. We assume contracts are correct, but some faults may require fixing the contracts themselves. We are working on ways to track down this type of faults.

Improved numerical constraints. For numerically constrained assertions, we only generates fixes involving minimal or maximal values. While this choice is empirically effective, more subtle faults may require the usage of non-extremal values.

Finding the best fix. We use very simple metrics to rank valid fixes; more research in this line may boost the quality of the ranking—and to determine what makes one fix “better” than another. One interesting approach is to look at known histories of changes and fixes to leverage this history for generating similar fixes.

7. CONCLUSIONS

In the past decade, automated debugging has made spectacular advances: First, we have seen the raise of methods to isolate failure causes automatically; then, statistical methods were devised that highlight likely failure locations. Now, we have reached a state where “automated debugging” truly deserves its name: we can actually attempt to automatically generate fixes.

Our experiences with AutoFix-E show that this goal is within reach: out of 42 faults found by AutoTest in two widely used Eiffel libraries, AutoFix-E proposes successful fixes for 16 faults. Since the entire process is fully automatic, this means that in 16 cases, the programmer's load could be reduced to almost zero. Note that we say "almost zero" here, 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 a strong specification in terms of 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-E source code, and all data and results cited in this article, are available at:

`http://se.inf.ethz.ch/research/autofix/`

Acknowledgments. This work is funded by Deutsche Forschungsgemeinschaft, Ze509/4-1 and Hasler-Stiftung, Grant no. 2327 under the title "AutoFix—Programs that fix themselves". The concept of generating fixes from differences in passing and failing runs was conceived with Andreas Leitner. The authors thank Jocelyn Fiat and Alexander Kogtenkov for their help in assessing the quality of automatically generated fixes. Valentin Dallmeier provided helpful feedback on an earlier version of this paper.

8. REFERENCES

[1] R. Abraham and M. Erwig. Goal-directed debugging of spreadsheets. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 37–44, 2005.

[2] R. Abraham and M. Erwig. Test-driven goal-directed debugging in spreadsheets. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 131–138, 2008.

[3] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *IEEE Congress on Evolutionary Computation*, pages 162–168. IEEE, 2008.

[4] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Experimental assessment of random testing for object-oriented software. In *ISSTA '07*, pages 84–94, New York, NY, USA, 2007. ACM.

[5] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *International Conference on Automated Software Engineering.* IEEE, 2009.

[6] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[7] B. Demsky and M. C. Rinard. Automatic detection and repair of errors in data structures. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–95. ACM, 2003.

[8] ECMA International. *Standard ECMA-367. Eiffel: Analysis, Design and Programming Language.* 2006.

[9] http://freeelks.svn.sourceforge.net.

[10] B. Elkarablieh and S. Khurshid. Juzi: a tool for repairing complex data structures. In *International Conference on Software Engineering*, pages 855–858, 2008.

[11] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. on Soft. Eng.*, 27(2):99–123, 2001.

[12] http://sourceforge.net/projects/gobo-eiffel/.

[13] H. He and N. Gupta. Automated debugging using path-based weakest preconditions. In *Fundamental Approaches to Software Engineering*, volume 2984 of *LNCS*, pages 267–280, 2004.

[14] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta. BugFix: A learning-based tool to assist developers in fixing bugs. In *IEEE International Conference on Program Comprehension*, pages 70–79, 2009.

[15] L. L. Liu, B. Meyer, and B. Schoeller. Using contracts and Boolean queries to improve the quality of automatic test generation. In *Tests and Proofs*, volume 4454 of *LNCS*, pages 114–130, 2007.

[16] M. Z. Malik, K. Ghori, B. Elkarablieh, and S. Khurshid. A case for automated debugging using data structure repair. In *Automated Software Engineering.* IEEE, 2009.

[17] http://www.wolfram.com.

[18] W. Mayer and M. Stumptner. Evaluating models for model-based debugging. In *Automated Software Engineering*, pages 128–137. IEEE, 2008.

[19] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2nd edition, 1997.

[20] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf. Programs that test themselves. *IEEE Software*, pages 22–24, 2009.

[21] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically correcting memory errors with high probability. *Communications of the ACM*, 51(12):87–95, 2008.

[22] J. H. Perkins, S. Kim, S. Larsen, S. P. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. C. Rinard. Automatically patching errors in deployed software. In *22nd ACM Symposium on Operating Systems Principles*, pages 87–102. ACM, 2009.

[23] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *Symposium on Principles of Programming Languages*, pages 313–326, 2010.

[24] S. Staber, B. Jobstmann, and R. Bloem. Finding and fixing faults. In *13th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, volume 3725 of *LNCS*, pages 35–49. Springer, 2005.

[25] W. Weimer. Patches as better bug reports. In *International Conference on Generative Programming and Component Engineering*, pages 181–190, 2006.

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