DISS. ETH NO. 22477

Automatic Fixing of Programs with Contracts

A dissertation submitted to attain the degree of
DOCTOR OF SCIENCES of ETH ZURICH

presented by
YU PEI
Ph.D. in Science, Nanjing University, Nanjing, China
Bachelor of Engineering, Nanjing University, Nanjing, China

born on
December 24th, 1977

citizen of
China

accepted on the recommendation of
Prof. Dr. Bertrand Meyer, examiner
Prof. Dr. Andreas Zeller, co-examiner
Prof. Dr. Harald Gall, co-examiner
Dr. Manuel Oriol, co-examiner

2015

ACKNOWLEDGEMENTS

First and foremost I thank Bertrand Meyer for hiring me as a research assistant and for giving me the opportunity to do a PhD under his supervision at ETH Zürich. His support and supervision made the work presented in this thesis possible.

I also thank the co-examiners of this thesis, Prof. Dr. Andreas Zeller, Prof. Dr. Harald Gall and Dr. Manuel Oriol, who kindly made time in their busy schedules to co-referee this work.

Among the present and past members of the Chair of Software Engineering at ETH Zürich, and without following any particular order, I would like to thank: Carlo Alberto Furia and Yi Jason Wei for the insightful discussions, the valuable advices, and the enormous help; Julian Tschannen for being always supportive and willing to help; Marco Trudel for the countless activities and parties that are worth remembering; Nadia Polikarpova for the special birthday card; Sebastian Nanz for the friendly small talks; Martin Nordio for the tremendous Asadors; Andrey Rusakov for the basketball games; Michael Schill for the enjoyable BBQs; Alexey Kolesnichenko for the discussions about algorithms and programming tasks; and Claudia Günthart for all the help over the years. And then, Chris Poskitt, Đurica Nikolič, Georgiana Caltais, Jiwon Shin, Marco Piccioni, Chandrakana Nandi, Christian Estler, Scott West, Benjamin Morandi, Stephan van Staden, Michela Pedroni, Ilinca Ciupa, and Mei Tang for the great atmosphere in the group.

Special thanks to Christian Estler for helping me analyze the code bases used in evaluating ImpleFix, and to the participants to the SpeciFix user study: Alexey Kolesnichenko, Nadia Polikarpova, Andrey Rusakov, and Julian Tschannen.

I would also like to thank Denise Spicher from the Student Administration Office for always answering my questions promptly and with patience.

Lastly, I especially thank my family for everything they have done for me. Without the support and encouragement of my parents, my sister, and my brother-in-law, I simply could not have first started and then survived this journey at ETH. My dear wife, Qianting Huang, quit her job in China and followed me to Zürich. Her love, encouragement, patience, and support were undeniably the bedrock upon which the past several years of my life have been built.

INHALTSVERZEICHNIS

ABSTRACT

Debugging—the activity of finding and correcting errors in programs—is so everyday in every programmer's job that any improvement at automating even parts of it has the potential for a significant impact on productivity and software quality. While automation remains formidably difficult in general, the last few years have seen the first successful attempts at automatically generating fixes to errors in some situations. This thesis aims at advancing the techniques and tools for the automatic fixing of errors in object-oriented programs with contracts (a.k.a. assertions).

To this purpose, the thesis has developed techniques and a supporting tool, collectively called AutoFix, that programmers can use in their everyday development to generate fixes to errors in an automatic fashion. AutoFix relies on the presence of simple specification elements in the form of contracts (such as pre- and post-conditions) to provide high-quality fix suggestions and to enable automation of the whole debugging process: using contracts enhances the precision of dynamic analysis techniques for fault detection and localization, and for validating fixes.

AutoFix consists of three major parts: the ImpleFix technique which generates fixes to the program implementation, the SpeciFix technique which suggests fixes to the contracts, and the AutoFix tool which implements both ImpleFix and SpeciFix.

Both ImpleFix and SpeciFix are driven by a set of test cases exercising the routine where the fault occurs. ImpleFix employs various program analysis techniques like dynamic invariant inference, simple static analysis, and fault localization to produce a collection of candidate fixes that change the implementation; while SpeciFix infers dynamic invariants in passing tests to summarize abstract program behavior, and synthesizes weakening and strengthening changes to the contracts to avoid failing behaviors. The generated fixes are validated against a regression test suite and ranked by preferring the ones that are more relevant to the failure or lead to less restrictive contracts. In the experiments conducted to evaluate the techniques, ImpleFix and SpeciFix generated fixes that are genuine corrections of quality comparable to those competent programmers would write to 25% of the subject faults.

The AutoFix tool is integrated into the Eiffel Verification Environment and functions like a recommendation system that is capable of automatically finding bugs and suggesting fixes in the form of source-code patches: it exploits the AutoTest random testing framework to detect errors and prepare test cases, and applies the ImpleFix and SpeciFix techniques to generate candidate fixes to the errors.

In conclusion, this thesis provides an automatic and integrated solution to the fixing of errors in object-oriented programs with contracts, which can greatly reduce the programmer's debugging effort in many cases.

ZUSAMMENFASSUNG

Debugging — das Finden und Korrigieren von Fehlern in Programmen — ist für Programmierer so alltäglich, dass sich Verbesserungen und Automatisierungen dieses Prozesses sehr positiv auf die Produktivität und Softwarequalität auswirken können. Obwohl Automatisierung in diesem Gebiet schwierig ist, wurden in den letzten Jahren erste erfolgreiche Ansätze zum automatischen Erzeugung von Fixes für Programmfehler vorgestellt. Diese Dissertation versucht die Techniken und Werkzeuge für das automatische Korrigieren von Fehlern in objekt-orientierten Programmen, welche mit Contracts versehen sind, zu verbessern.

Wir präsentieren in dieser Dissertation sowohl eine Technik, als auch ein dazugehöriges Werkzeug, die gemeinsam als AutoFix bezeichnet werden. Programmierer können AutoFix in ihrer täglichen Arbeit benutzen, um Korrekturen von Fehlern automatisch zu erzeugen. AutoFix stützt sich auf einfache Spezifikationselemente in Form von Contracts (z.B. Vor- und Nachbedingungen), um qualitativ hochwertige Vorschläge für Korrekturen zu erzeugen und eine Automatisierung des gesamten Debugging-Prozesses zu ermöglichen: Contracts erlauben dabei die Präzision der dynamischen Analysetechniken zur Erkennung und Lokalisierung von Fehlern zu erhöhen und der automatische Bewertung generierter Fixes zu verbessern.

AutoFix besteht aus drei Hauptteilen: der ImpleFix Technik, welche Korrekturen für Programmcode erzeugt; der SpeciFix Technik, welche Korrekturen von Contracts vorschlägt; und dem AutoFix Werkzeug, das sowohl ImpleFix als auch SpeciFix implementiert.

Sowohl ImpleFix als auch SpeciFix nutzen eine Gruppe von Tests, welche den Fehler einer Routine aufzeigen. ImpleFix benutzt verschiedene Analysetechniken wie dynamische Inferenz von Invarianten, einfache statische Analyse und Fehlerlokalisierung, um eine Sammlung von Kandidaten-Fixes zu produzieren, welche die Implementierung der Routine verändern. SpeciFix leitet aus erfolgreichen Tests dynamische Invarianten ab, um abstraktes Programmverhalten zusammenzufassen, und erzeugt abschwächende oder verstärkende Änderungen der Contracts, um fehlerhaftes Verhalten zu vermeiden. Die erzeugten Fixes werden mithilfe einer Regressionstestsuite validiert und gewichtet, wobei die Fixes, wel-

che relevanter für den Programmfehler oder zu weniger einschränkenden Contracts führen, bevorzugt werden. In den Experimenten, welche zur Evaluation der Techniken durchgeführt wurden, haben ImpleFix und SpeciFix in 25% der Fälle Fixes generiert, welche vergleichbar mit Fixes von erfahrenen Programmieren sind.

Das AutoFix Werkzeug ist in die Eiffel Verifikationsumgebung integriert und funktioniert wie ein Empfehlungssystem, das automatisch Fehler findet und Fixes in Form von Quellcode-Patches vorschlägt: es benutzt das AutoTest Test-Framework um Fehler zu erkennen und Tests vorzubereiten, und verwendet die ImpleFix und SpeciFix Techniken um Kandidaten-Fixes für Fehler zu erzeugen.

Zusammengefasst bietet diese Arbeit eine automatische und integrierte Lösung für das Korrigieren von Fehlern in objekt-orientierten Programmen mit Contracts, die den Debuggingaufwand für Programmierer erheblich reduzieren kann.

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

CHAPTER 1

OVERVIEW

Programs have errors, and the programmer's ever recommencing fight against errors involves two tasks: finding errors; and correcting them.

Depending on when they can be detected, errors fall into two categories in general: *static errors* are caught when the code is compiled and they prevent the programs from running; *dynamic errors* will, however, lead to unexpected final results or program behaviors, e.g. premature termination. Dynamic errors are indeed discrepancies between the program implementation (what a program actually does) and the program specification (what the program is supposed to do).

Programmers are now used to the help from modern Integrated Development Environments (IDEs) in fighting against static errors: incremental compilation is able to spot such errors instantly during development, then the IDE will propose viable actions to correct the errors. For example, Eclipse constructs a list of quick fixes when it detects a static error, indicating actions that can be undertaken to repair the error. A programmer then needs to review these actions and select the appropriate ones to apply. In this way, the manual effort needed to fix static errors is greatly reduced.

Compared with static errors, dynamic errors are more challenging to find and to fix, yet no similar tool help in handling dynamic errors is available to programmers: although techniques to *find* dynamic errors automatically are now becoming increasingly available and slowly making their way into industrial practice [15, 41, 88], the task of *correcting* dynamic errors has remained largely a manual effort due to its inherent difficulties.

The purpose of this thesis is to advance the techniques and tools for the automatic fixing of dynamic errors in object-oriented programs with contracts (a.k.a. assertions). Most modern programming languages are based on the object-oriented paradigm, and some of them support contracts, either as part of the lan-

guage (e.g. Eiffel and Spec#) or as an extension (e.g. JML for Java and code contracts for C#), for specifying the mutual conditions as well as the obligations among software elements. Contracts provide a specification of correct behavior that can be used not only to detect dynamic errors automatically [70] but also to suggest corrections.

To this purpose, the thesis has developed techniques and a supporting tool to automatically generate corrections to dynamic errors. The current implementation of the supporting tool is integrated in the open-source Eiffel Verification Environment (EVE) [39]—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).

In the rest of this thesis, we refer to "dynamic errors" as simply "errors", and we use the terms "error", "fault", and "bug" interchangeably.

1.1 Main Results

The contributions of this thesis include the following:

- The ImpleFix technique to generate implementation fixes to program faults. The technique combines various program analysis techniques such as dynamic invariant inference, simple static analysis, and fault localization. ImpleFix is driven by a set of test cases that exercise the routine (method) where the fault occurs, and produces a collection of suggested fixes that change the implementation, ranked according to a heuristic measurement of relevance.

  An extensive experimental evaluation, detailed in Section 4.2, showed ImpleFix is able to automatically suggest *quality fixes* to program faults with *high success rate*. Fix quality is an aspect largely neglected [71] in the research on program repair. ImpleFix's work is an exception, as we introduced the notion of proper fix [103] to characterize those that correct the faults under consideration while not introducing new faults. In our experiments, ImpleFix successfully suggested proper fixes to 51 out of 204 faults from four code bases.

- The SpeciFix technique, which produces specification fixes to program faults and complements ImpleFix. SpeciFix works on the same type of input as ImpleFix. It infers dynamic invariants in passing runs to summarize abstract program behavior, and it synthesizes weakening and strengthening changes to the contracts that do not violate the invariants. SpeciFix ranks the candidate fixes by preferring those that are less restrictive.

Program faults are discrepancies between the implementation and the specification, and the cause of a fault may be on either the implementation side or the specification side. To the best of our knowledge, all the previous research on program repairing has aimed to correct faults by changing the implementation. SpeciFix tackles the problem from a different angle and suggests specification fixes to program faults. In an experiment involving 44 faults from production software, SpeciFix produced proper fixes to contracts for 11 (or 25%) of the faults.

- The AutoFix tool, which combines ImpleFix and SpeciFix, and is capable of automatically finding bugs and suggesting fixes in the form of source-code patches.

  The AutoFix tool exploits the AutoTest random testing framework to find faults and generate test cases for fixing, therefore the only input it expects from the user is a program annotated with the same contracts that programmers using a contract-equipped language normally write [91, 38]. The AutoFix tool has been integrated into EVE, which makes automatic program fixing readily accessible to programmers in their everyday development.

  In its typical use scenario, the AutoFix tool runs in the background or during work interruptions, displaying fix suggestions as they become available. Such usage is feasible since both ImpleFix and SpeciFix require limited computational resources to produce the fixes: the average time per fix was below 20 minutes on commodity hardware in our experiments. Such performance compares favorably with the other state-of-the-art repair techniques based on dynamic analysis: for example, GenProg reported an average time cost of nearly 100 hours for each fix in a recent experiment [54].

  In the rest of this thesis, we refer to the ImpleFix technique, the SpeciFix technique, and the AutoFix tool collectively as AutoFix.

AutoFix is the result of a joint project, also called AutoFix, between the Chair of Software Engineering at ETH Zurich and the Software Engineering Chair at Saarland University. AutoFix builds on the previous work on automatic testing at ETH, leading to the AutoTest tool [70], and on the work on automatic debugging at Saarland, leading in particular to the Pachika tool [28].

1.2 Structure

From a user's perspective, the two most important characteristics of an automatic fixing tool are its capability to produce high quality fixes and the easiness of using the tool. Chapter 2 demonstrates such capability of AutoFix with two example

faults and describes its integration into EVE which enables the programmers to use automatic fixing easily and routinely in their development.

The dynamic analysis techniques that AutoFix exploits to localize faults and synthesize fixes are based on concepts like expressions and predicates, execution traces, contracts and correctness, and test cases. Chapter 3 summarizes these concepts.

As an appropriate way to correct an error may require changes to the program implementation or the specification, AutoFix generates candidate fixes of both types: Chapter 4 describes the ImpleFix technique for generating fixes to the implementation and Chapter 5 the SpeciFix technique for producing fixes that change the contracts. Both chapters also relate the experimental evaluations of the techniques.

Chapter 6 presents related work and compares it with our contribution. Finally, Chapter 7 draws conclusions on the work done and suggests future work.

CHAPTER 2

AUTOFIX IN ACTION

AutoFix is designed to automate the process of program fixing as much as possible. Its desirability in practice depends largely on both the quality of the proposed candidate fixes and the easiness of using the tool. This chapter demonstrates AutoFix's capability to propose high quality fixes with two example faults (Section 2.1), which originate from the real world code and were included in the experiments in Sections 4.2 and 5.2, and describes the integration of AutoFix into EVE, which makes automatic fixing easily accessible to programmers in their daily work.

The discussion here shows the results from the AutoFix user's perspective; it does not explain the technology that underlies these results. The following chapters will present that technology.

2.1 Two example faults

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

2.1.1 Moving Items in Sorted Sets

The first fault comes from class *TWO_WAY_SORTED_SET*, which is the standard Eiffel implementation of sets using a doubly-linked list. Listing 2.1 outlines features (members) of the class, some annotated with their pre- (**require**) and postconditions (**ensure**). [1] As pictured in Figure 2.1, 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

[1] 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
```

Listing 2.1: Some features of class *TWO_WAY_SORTED_SET*.

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.

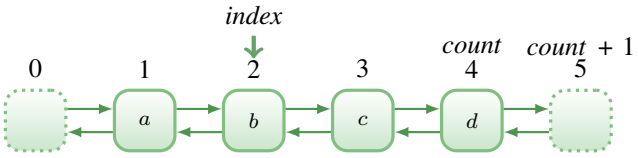


Figure 2.1: A doubly-linked list implementing *TWO_WAY_SORTED_SET*. The cursor *index* is on position 2. The elements are stored in positions 1 to 4, whereas positions 0 (*before*) and 5 (*after*) mark the list's boundaries. *count* denotes the number of stored elements (i.e., four).

Listing 2.1 also shows the complete implementation of routine (method) *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 Figure 2.1), *move_item* (*d*) changes the list to $\langle a, d, b, c \rangle$. *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*.

2.1.1.1 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*.

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; and the following call to *go_i_th* triggers a precondition violation, as the feature requires the argument to be between 0 and *count* + 1 (line 23).

This fault is 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

```
if idx > index then
  idx := idx − 1
end
```

Listing 2.2: Correction of the error in *move_item* automatically generated by AutoFix.

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.[2]

2.1.1.2 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 Listing 2.2, 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 in Listing 2.2 before the call to *go_i_th* on line 41 in Listing 2.1. 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 to formalize its intended behavior.

2.1.2 Duplicating Circular Lists

The second example targets a bug of routine *duplicate* in class *CIRCULAR*, which implements circular lists based on arrays.

To understand the bug, Listing 2.3 illustrates a few details of *CIRCULAR*'s API. Lists are numbered from index 1 to index *count* (an attribute denoting the list

[2]Recent work [92, 90, 93] has led to new versions of the libraries with strong (often complete) contracts, capturing all relevant postcondition properties.

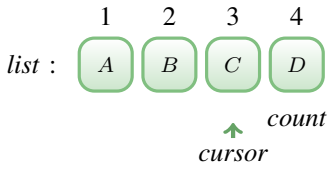


Figure 2.2: A circular list of class *CIRCULAR*: the internal *cursor* points to the element C at index 3.

length), and include an internal *cursor* that may point to any element of the list. Routine *duplicate* takes a single integer argument *n*, which denotes the number of elements to be copied; called on a list object *list* , it returns a new instance of *CIRCULAR* with at most *n* elements copied from *list* starting from the position pointed to by *cursor*. Since we are dealing with circular lists, the copy wraps over to the first element. For example, calling *duplicate* (3) on the *list* in Figure 2.2 returns a fresh list with elements $\langle C, D, A \rangle$ in this order.

The implementation of *duplicate* is straightforward: it creates a fresh *CIRCULAR* object **Result** (line 55 in Listing 2.3); it iteratively copies *n* elements from the current list into **Result**; and it finally returns the list attached to **Result**. The call to the creation procedure (constructor) *make* on line 55 allocates space for a list with *count* elements; this is certainly sufficient, since **Result** cannot contain more elements than the list that is duplicated. However, *CIRCULAR*'s creation procedure *make* includes a precondition (line 50 in Listing 2.3) that only allows allocating lists with space for at least one element (**require** $m \geq 1$). This sets off a bug when *duplicate* is called on an empty list: *count* is 0, and hence the call on line 55 triggers a violation of *make*'s precondition. Testing tools such as AutoTest

```
47 class CIRCULAR [G]
48
49   make (m: INTEGER)
50       require    m ≥ 1
51       do ... end
52
53   duplicate (n: INTEGER): CIRCULAR [G]
54       do
55           create Result.make (count)
56           ...
57       end
58
59   count: INTEGER  -- Length of list
```

Listing 2.3: Some implementation details of *CIRCULAR*.

```
make (m: INTEGER)
  require    m ≥ 1

duplicate  (n: INTEGER):
        CIRCULAR [G]

  do
    if  count  > 0 then
        create  Result.make (count)
    else
        create  Result.make (1)
    end
```

(a) Patching the implementation.

```
make (m: INTEGER)
  require    m ≥ 1

duplicate  (n: INTEGER):
        CIRCULAR [G]
  require   count  > 0
  do

    create  Result.make (count)
```

(b) Strengthening the specification.

```
make (m: INTEGER)
  require    m ≥ 0

duplicate  (n: INTEGER):
        CIRCULAR [G]

  do

    create  Result.make (count)
```

(c) Weakening the specification.

Listing 2.4: Three different fixes for the bug of Listing 2.3. Changed or added lines are highlighted.

detect this bug automatically by providing a concrete test case that exposes the discrepancy between implementation and specification.

How should we fix this bug? Listing 2.4 shows three different possible repairs, all of which AutoFix can generate completely automatically. An obvious choice is patching *duplicate* 's implementation as shown in Listing 2.4a: if *count* is 0 when *duplicate* is invoked, allocate **Result** with space for *one* element; this satisfies *make*'s precondition in all cases.

This fix to the implementation is acceptable, since it makes *duplicate* run correctly, but it is not entirely satisfactory: *CIRCULAR*'s implementation looks perfectly adequate, whereas the ultimate source of failure seems to be incorrect or inadequate *specification*. A straightforward fix is then adding a precondition to *duplicate* that only allows calling it on non-empty lists. Listing 2.4b shows such a fix, which *strengthens duplicate* 's precondition thus invalidating the test case exposing the bug. The strengthening fix has the advantage of being textually simpler than the implementation fix, and hence also probably simpler for programmers to understand. However, both fixes in Listings 2.4a and 2.4b are partial, in that they remove the source of faulty behavior in *duplicate* but they do not prevent similar faults—deriving from calling *make* with $m = 0$—from happening. A more critical issue with the specification-strengthening fix in Listing 2.4b is that it may break clients of *CIRCULAR* that rely on the previous weaker precondition.[3] There are cases where strengthening produces the most appropriate fixes; in the running example, however, strengthening arguably is not the optimal strategy.

[3]Note that this strengthening does not introduce new bugs; it just shifts the responsibility for the fault from *duplicate* to its clients.

A look at *make*'s implementation (not shown in Listing 2.3) would reveal that the creation procedure's precondition $m \geq 1$ is unnecessarily restrictive, since the routine body works as expected also when executed with $m = 0$. This suggests a fix that *weakens make*'s precondition as shown in Listing 2.4c. This is arguably the most appropriate correction to the bug of *duplicate* : it is very simple, it fixes the specific bug as well as similar ones originating in creating an empty list, and it does not invalidate any clients of *CIRCULAR*'s API. AutoFix generates both specification fixes in Listings 2.4b and 2.4c but ranks the weakening fix higher than the strengthening one. More generally, AutoFix outputs specification-strengthening fixes only when they do not introduce bugs in available tests, and it always prefers the least restrictive fixes among those that are applicable.

2.2 The AutoFix Tool

The AutoFix tool implements the automatic fixing techniques and provides a user-friendly interface for these techniques so that they are easily accessible to programmers. The AutoFix tool requires a program to be debugged as the only user input: although the underlying fixing techniques take a set of passing and failing test cases as input, the tool relieves programmers of the burden of preparing such tests by exploiting AutoTest to automatically detect bugs and generate tests for fixing; the AutoFix tool requires only a little effort to set up and launch, yet when successful it reports faults together with ready-to-apply candidate fixes, making debugging much easier: the candidate fixes involve only simple changes to the program and the changes are presented side by side with the original source for better readability; if a candidate fix to the implementation is appropriate, then the programmer just needs to click a button to apply it to the code.

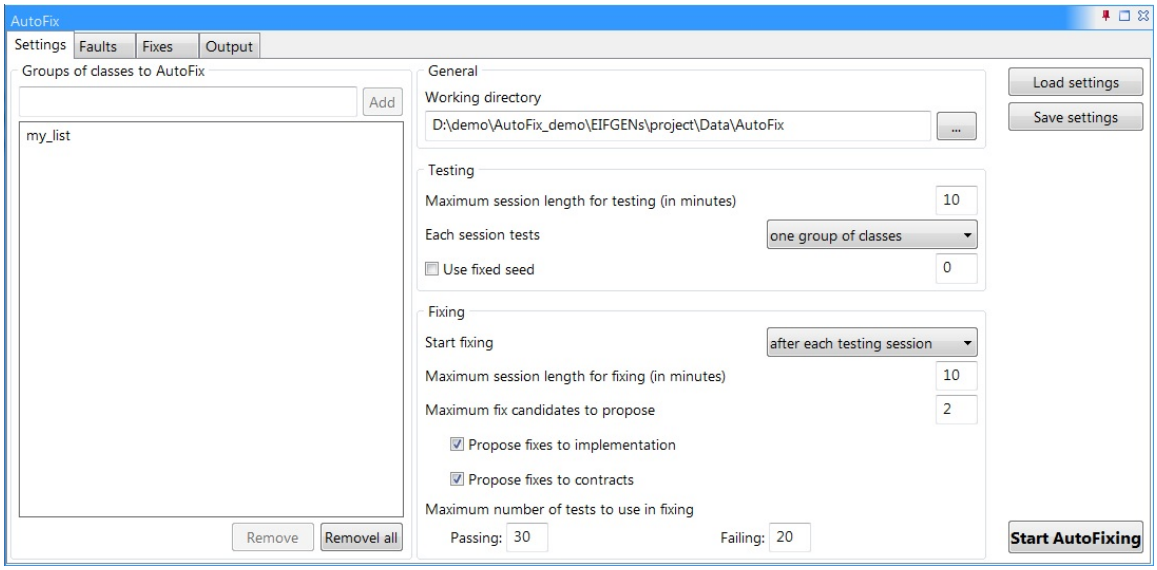
This section describes the AutoFix tool user interface and explains how the tool can be used easily by programmers in the development process on a daily basis through a typical use case scenario.

2.2.1 The Graphical User Interface

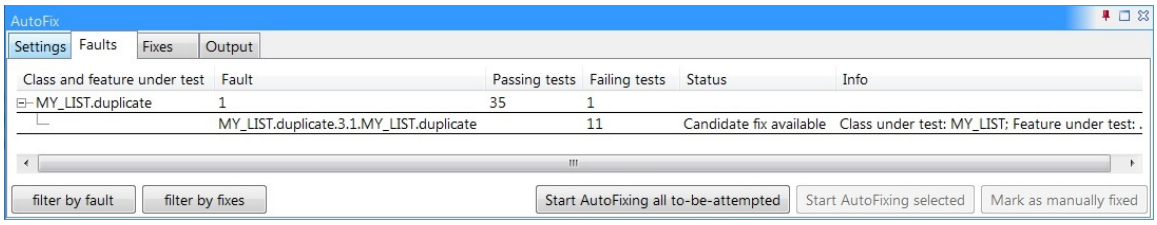
The graphical user interface (GUI) of the AutoFix tool consists of four panels: the Settings panel for the tool configuration, the Faults panel and the Fixes panel for providing a centralized view of all the detected faults and generated candidate fixes, respectively, and the Outputs panel for intermediate tool outputs.

The Settings panel (Figure 2.3a) provides a means for the user to configure the tool behavior easily. The meanings of the settings, and the restrictions on their values when applicable, are as follows:

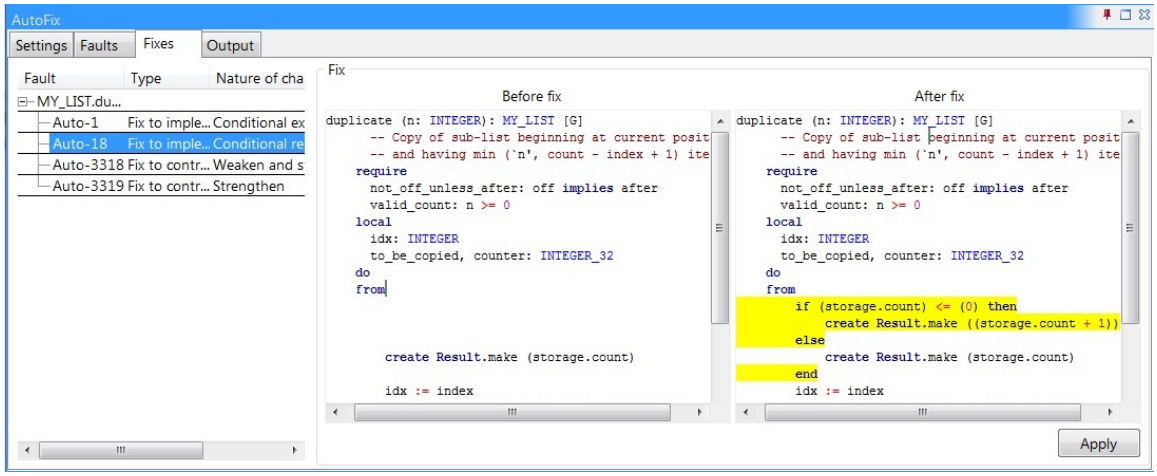
1. Groups of classes to AutoFix. Some classes can be tested more thoroughly



(a) The Settings panel



(b) The Faults panel



(c) The Fixes panel

Figure 2.3: From top to bottom: the Settings panel, the Faults panel and the Fixes panel of the AutoFix tool.

if they are tested together, e.g. when their interfaces strongly depend on each other, such classes can be specified to AutoFix in groups. Class groups do not need to disjoint with each other.

2. Working directory. This is where all the intermediate files and results will be stored during automatic testing and fixing.

3. Maximum session length for testing (in minutes). This positive integer defines the lengths of the testing sessions using AutoTest.

4. What to test in each session? Each testing session can be configured to test: one class, one group of classes, or all classes.

5. Use fixed seed. Although the automatic fixing process is deterministic in that the same input test cases will always produce the same candidate fixes, the AutoTest tool exercises random testing: it uses a seed to generate a sequence of random numbers to control how testing proceeds. By default, AutoTest uses a new seed for each testing session and therefore different testing sessions produce different testing results, which in turn lead to different fixing sessions. When reproducible fixing sessions are desirable, a fixed seed for AutoTest can be provided here.

6. When to start fixing? Program fixing can be started automatically after each testing session, after all testing sessions, or manually.

7. Maximum session length for fixing (in minutes). This non-negative integer defines the maximum length of a fixing session. Value 0 means no time limit is set for fixing sessions.

8. Maximum fix candidates to propose. This positive integer decides the maximum number of candidate fixes the tool will produce for each fault. The candidate fixes may target either the program implementation or the contracts.

9. Maximum number of tests to use in fixing. The non-negative numbers determine how many tests will be used as input in each fixing session. Value 0 means all available tests should be used.

The most important functionalities of the AutoFix tool are organized into two **views**: the Faults view and the Fixes view, corresponding to the Faults panel (Figure 2.3b) and the Fixes panel (Figure 2.3c) respectively.

The Faults view lists all the faults detected by AutoTest, with detailed information about each fault like the fault signature, the numbers of corresponding passing and failing tests, and the status of the fault (e.g. whether the fault has

been fixed or not). For example, Figure 2.3b shows the details about a fault: from its id *MY_LIST.duplicate* .3.1. *MY_LIST.duplicate* we know the fault triggered a failure during testing routine *duplicate* from class *MY_LIST* (first occurrence of *MY_LIST.duplicate*), and the failure was a precondition violation (exception code 3) when executing the first instruction (instruction index 1) of *MY_LIST.duplicate* (second occurrence of *MY_LIST.duplicate*); AutoTest has generated 40 passing tests for routine *duplicate* and 13 failing tests revealing the very same fault, which can later be used for fixing the fault.

The Fixes view enumerates all the proposed candidate fixes to each fault. When a particular fix is selected, relevant code snippets before and after applying the fix will be shown, with their differences highlighted, to facilitate fix review. For example, the fix with id *Auto*−18, as shown in Figure 2.3c, requires adding an **if** conditional around an instruction so that a different operation is executed when *storage* .*count* is less than or equal to 0. Also in this view, once the user decides an implementation fix is good enough, she only needs to first select the fix and then click on the button "Apply" to deploy the fix and correct the corresponding fault.

2.2.2 A Typical Use Scenario

The AutoFix tool can run on every program that compiles, and therefore is applicable in different scenarios during both software implementation and maintainance. Consider the following typical use scenario of AutoFix from the perspective of a nondescript user named Alice.

As Alice checks in for work today, she finds out that the latest version of EiffelStudio—the IDE she normally uses for development—includes a new pane with a tool called AutoFix. AutoFix's interface (Figure 2.3) looks familiarly similar to other tools already available in EiffelStudio that Alice routinely used. In fact, the only required input to the AutoFix pane is simply the name of one class to be analyzed.

Later during the day, Alice decides to give AutoFix a try. Class *MY_LIST* she's been working on during the morning now includes implementations of the main public features; this seems a good time to start testing. Since it's almost noon, Alice launches AutoFix on *MY_LIST* with default settings and leaves for lunch.

When she's back after one hour, Alice sees that automatic testing (ran by AutoFix using contracts as oracles) has found a bug in *MY_LIST*'s copy method *duplicate* (Figure 2.3b). In a different tab (Figure 2.3c), AutoFix lists four different fix suggestions: two of them change the code, whereas the other two change the available contracts.

Using AutoFix's diff view, Alice quickly inspects the bug and the suggested fixes; she immediately finds out that the bug is triggered when *duplicate* is called

on an empty list. All the four fixes have been validated against the available tests, but Alice singles out two of them as fully satisfactory: one creates and returns an empty list as a special case; another one relaxes the precondition of a constructor that turned out to work correctly even in the case of empty lists. Since relaxing the precondition would change the API contracts, which were previously agreed to by all members of the development team, Alice opts for deploying the implementation fix that handles the empty list case separately. This requires one click.

AutoFix has helped Alice find and correct an error in the codebase in a matter of minutes. Now, she can continue extending the implementation of *MY_LIST* with more confidence in the correctness of existing code. Alice plans to start another, longer AutoFix session when she'll leave in the evening.

CHAPTER 3

AUTOFIX CONCEPTS

AutoFix employs dynamic analysis techniques to localize faults and synthesize fixes (see Chapters 4 and 5). Such techniques are based on concepts like expressions and predicates, execution traces, contracts and correctness, and tests. This chapter summarizes these concepts and lays the foundation for the automatic fixing techniques presented in the following two chapters.

3.1 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:

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

3.1.1 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 Figure 2.1), 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 argu-

mentless, 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 [60, 28] showed the effectiveness of Boolean queries as a guide to partitioning the state space for testing and other applications.

3.1.2 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 Figure 2.1: $E_{\mathit{before},\ \mathit{index}\ >1}$ 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* [91]: $\mathbb{E}_{r,c}$ 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, $\mathbb{E}_{\mathit{before},\mathit{index}\ >1} = E_{\mathit{before},\mathit{index}\ >1}$ because all the expressions in $E_{\mathit{before},\mathit{index}\ >1}$ 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 3.1.1);

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,[1] and every comparison operator $\sim$ in $\{=, <, \leq\}$;

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

In the example, $B_{\mathit{before},\mathit{index}\ >1}$ contains **Result** and **not Result**, since **Result** has Boolean type; the comparisons *index* <0, *index* ≤ 0, *index* = 0, *index* $\neq 0$,

[1] The constant 0 is always included because it is likely to expose relevant cases.

index ≥ 0, and *index* > 0; and the same comparisons between *index* and the constant 1.

3.1.3 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 (Figure 2.1); 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 [30]: 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.

3.2 Execution Traces of Routines

Expressions and predicates enable us to focus on the interesting aspects of the program state at any point during the execution of routines. To be able to understand the real cause for a fault, we also need to examine how the program state evolves, that is how the instructions at different locations transit the program from one state to another. We model such evolvement using execution traces.

Let $\mathcal{S}$ be a set of program states, abstracted using expressions and predicates as defined in Section 3.1, and $\mathcal{L}$ the set of program locations. For simplicity of presentation, we assume that each routine r has two boundary locations: execution of r always starts from its *entry location* $l_{\overline{r}} \in \mathcal{L}$ and returns, if ever, from its *exit*

location $l_{\underline{r}} \in \mathcal{L}$; instructions at routine boundary locations are always the null operation NOP.

Given a call of routine r with actual arguments $a_1, \ldots, a_n$ on a target object a_0, written $\tau = a_0.r(a_1, \ldots, a_m)$, its execution starts from an initial state s_0 at location $l_0 = l_{\overline{r}}$, and defines uniquely a *trace* ρ_τ as the sequence

$$\rho_\tau = s_0\, l_0\, s_1\, l_1\, \cdots\, s_{n-1}\, l_{n-1}\, s_n\, l_n\, \cdots \tag{3.1}$$

of states and locations, where $s_i \in \mathcal{S}$ and $l_i \in \mathcal{L}$ ($i \geq 0$). A triple $s_i l_i s_{i+1}$ in ρ_τ denotes that the execution reaches location l_i in state s_i, and the instruction at location l_i transits the program state to s_{i+1}. If l_i is the entry location of a routine r_k, then s_i is the *pre-state* of the routine call; if l_i is the exit location of routine r_k, then s_i is the *post-state* of the routine call. The sequence

$$\kappa_\tau = r_0\, r_1\, \cdots\, r_{n-1}\, r_n\, \cdots \tag{3.2}$$

containing only routine names of the boundary locations in ρ_τ is the *call sequence* determined by τ. Since τ is a call to r at the outmost level, $r_0 = r$.

Consider feature *duplicate* in Listing 2.3, the call *emp.duplicate* (1) determines the trace $x_0\, l_{\overline{duplicate}}\, x_1\, l_{line55}\, x_2\, l_{\overline{make}}$ where x_0 is the initial state, $x_1 = x_0$, and x_2 is the state when calling *make* on line 55 in Listing 2.3; the test terminates then with a contract violation. Another call *list . duplicate* (3) determines the call sequence *duplicate make make duplicate* .

3.3 Contracts and Correctness

AutoFix works on Eiffel classes equipped with *contracts* [69]. 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 thesis). Each assertion consists of one or more *clauses*, implicitly conjoined and usually displayed on different lines; Consider for example routine *move_item* in Listing 2.1, its precondition has two clauses: $v \neq$ **Void** on line 30 and *has*(v) on line 31. We denote by P_r and Q_r the pre- and postcondition of a routine r.

Given an assertion A and a program state $s \in \mathcal{S}$, we say that A *holds* at s (or, equivalently, that s satisfies A) if A evaluates to **True** under state s; if this is the case, we write $s \models A$. Since contracts are executable, we can evaluate any assertion at any program state reached during a concrete execution.

Contracts provide an operational criterion to classify routine calls into invalid, successful, and failing. A call $\tau = a_0.r(a_1, \ldots, a_m)$ is *valid* if the initial state s_0

of the trace ρ_τ is such that it satisfies r's precondition, that is $s_0 \models P_r$; otherwise τ is *invalid*. A valid routine call τ is *successful* if it returns and, for every $j (j > 0)$, location l_j in τ's trace ρ_τ satisfies the following: if $l_j = l_{\overline{r_j}}$ for a routine r_j then $s_j \models P_{r_j}$, and if $l_j = l_{\underline{r_j}}$ then $s_j \models Q_{r_j}$. In words, every nested call performed during the computation of r starts in a state that satisfies the called routine's precondition and terminates in a state that satisfies the called routine's postcondition when it returns. A valid routine call is *failing* if it is not successful, that is if it eventually reaches a state that violates some assertion; the violation terminates routine execution.

Take routine *duplicate* in Listing 2.3 as an example, the call *emp.duplicate* (1) is valid but failing: the nested call to *make* does not satisfy *make*'s precondition *m* ≥ 1 on line 50 in Listing 2.3 because *count* = 0 < 1 in an empty list; while the other call *list . duplicate* (3) is successful because the execution of *duplicate* terminates without violating any contract (and produces the correct result).

3.4 Tests

In this work, a test case t, or just "test", is simply a call to a routine r, and t is valid, passing, or failing, if and only if the call to r is valid, successful, or failing, respectively. An invalid test case for routine r does not tell us anything about r's correctness, since every invocation of r should satisfy r's precondition to be acceptable; a failing test case t reveals a fault in routine r; conversely, a passing test case documents a legitimate usage of routine r with respect to its specification. As we forcibly terminate tests that are still running after a timeout, all traces of tests are finite. Two failing test cases t_1, t_2 identify the *same fault* if their traces ρ_1, ρ_2 end with the same location f and violate the same assertion c.

Every session of automated program fixing takes as input a set T of test cases, partitioned into sets P (passing) and F (failing), and each session targets a single specific fault. When we want to make the targeted fault explicit, we write T_r, P_r, and $F_r^{f,c}$. For example, $F_{move\_item}^{42,\textbf{not}\ before}$ denotes a set of test cases all violating *put_left* 's precondition at line 42 in *move_item* (Listing 2.1).

The fixing algorithms described in Chapter 4 and Chapter 5 are independent of whether the test cases T are generated automatically or written manually. The experiments discussed in both chapters use the random testing framework AutoTest [70]. 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 to the implementation.

CHAPTER 4

FIXING THE IMPLEMENTATION

At the core of AutoFix are the techniques for automatically suggesting fixes to faults. As a bug manifests a discrepancy between specification and implementation, such techniques should be able to suggest fixes with respect to both the implementation and the contracts.

This chapter presents the ImpleFix technique of AutoFix to generate corrections to program implementations completely automatically. This part of the research is described in [86, 87]. The SpeciFix technique that aims at automatically proposing fixes to contracts is described in Chapter 5.

Section 4.1 explains the ImpleFix 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 4.2 discusses the experimental evaluation, including a detailed statistical analysis of numerous important measures.

4.1 How ImpleFix Generates Fixes to the Implementation

Figure 4.1 summarizes the steps of ImpleFix processing, from program to fix. The following subsections give the details.

ImpleFix takes a program with contracts as input, and it first employs AutoTest to generate test cases for the program. Each generated failing test exposes a fault in the program, which can be characterized by a program location f and by a violated contract clause c (Section 3.4); the presentation in this section leaves f and c implicit whenever clear from the context. To generate fixes to a fault, ImpleFix examines the execution of a group of tests, including the failing tests exposing the same fault and the passing ones testing the same routine as the se-

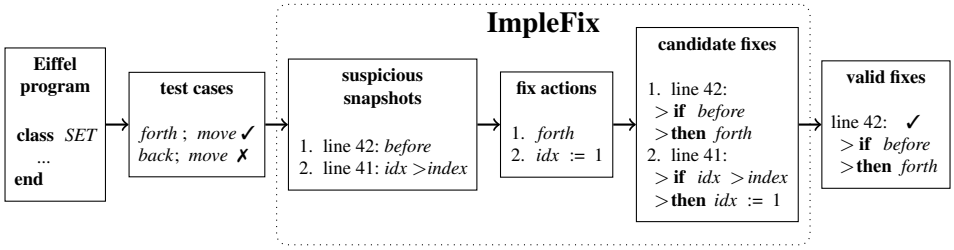


Figure 4.1: How ImpleFix generates fixes to the program implementation. Given an Eiffel program with contracts (Section 3.3), we generate passing and failing test cases that target a faulty routine (Section 3.4). By comparing the program state during passing and failing runs, ImpleFix identifies *suspicious snapshots* (Sections 4.1.1–4.1.2) that denote likely locations and causes of failures. For each suspicious snapshot, ImpleFix generates *fix actions* (Section 4.1.3) that can change the program state of the snapshot. Injecting fix actions into the original program determines a collection of *candidate fixes* (Section 4.1.4). The candidates that pass the regression test suite are *valid* (Section 4.1.5) and output to the user.

lected failing tests. The notion of *snapshot* (described in Section 4.1.1) is the fundamental abstraction for characterizing and understanding the behavior of the program in the test cases; ImpleFix 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 4.1.2) boils down to ranking snapshots according to a combination of static and dynamic analyses that search for the origins of faults.

Once ImpleFix has decided where to modify the program, it builds a code snippet that changes the program behavior at the chosen location. ImpleFix synthesizes such *fix actions*, described in Section 4.1.3, by combining the information in snapshots with heuristics and behavioral abstractions that amend common sources of programming errors.

ImpleFix injects fix actions at program locations according to simple conditional schema; the result is a collection of *candidate fixes* (Section 4.1.4). The following *validation* phase (Section 4.1.5) determines which candidate fixes pass all available test cases and can thus be retained.

In general, ImpleFix builds several valid fixes for the same fault; the valid fixes are *ranked* according to heuristic measures of "quality" (Section 4.1.6), so that the best fixes are likely to emerge in top positions.

4.1.1 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*[1]: 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, $\langle 31, v = \textbf{Void}, \textbf{False} \rangle$ describes that the predicate $v =$ **Void** evalutes 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 3.4, ImpleFix constructs a set $\mathsf{snap}(T_r)$ of snapshots. The snapshots come from two sources: invariant analysis (described in Section 4.1.1.1) and enumeration (Section 4.1.1.2).

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 $\ell \in \mathsf{loc}(t)$, $[\![e]\!]_t^\ell$ is the value of e at ℓ in t, if e can be evaluated at ℓ (otherwise, $[\![e]\!]_t^\ell$ is undefined).

4.1.1.1 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. ImpleFix uses Daikon [35] to infer invariants that characterize the *passing* and *failing* runs; their difference determine some snapshots that highlight possible failure causes. [2]

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 $F_r^{f,c} \subseteq T_r$. ImpleFix uses only invariants built out of publicly visible predicates in $\mathbb{P}_{r,c}$. The predicates in $\Pi = \{p \mid p \in \phi_\ell \text{ and } \neg p \in \pi_\ell\}$ characterize potential causes of errors, as Π contains predicates that hold in failing runs but not in passing runs.[3] Correspondingly, the set $\mathsf{snap}(T_r)$ includes all components

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

[1] In previous work [87], we used the term "component" instead of "snapshot".

[2] 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 3.1.

[3] Since the set of predicates used by ImpleFix is closed under complement (Section 3.1), Π is equivalently computed as the negations of the predicates in $\{p \mid p \in \pi_\ell \text{ and } \neg p \in \phi_\ell\}$.

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 [58]. The dynamic analysis described in Section 4.1.2.2 would assign the highest dynamic score to snapshots whose predicates correspond to the deterministic bug predictors in cooperative bug isolation.

4.1.1.2 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 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 2.1.1), the snapshots include, among many others, $\langle 34, v = \textbf{Void}, \textbf{False} \rangle$ (every execution has $v \neq \textbf{Void}$ when it reaches line 34) and $\langle 41, idx > index, \textbf{True} \rangle$ (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 4.2), 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 4.1.2.3) typically targeted only one or two locations ℓ with different predicates p.

4.1.2 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 ImpleFix computes a number of heuristic measures for each snapshot, described in the following subsections; these include simple syntactic measures such as the distance between program statements (Section 4.1.2.1) and metrics based on the runtime behavior of the program in the passing and failing tests (Section 4.1.2.2).

The various measures are combined in a *ranking* of the snapshots (Section 4.1.2.3) 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.

4.1.2.1 Static analysis

The static analysis performed by ImpleFix 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 [4, 72]. ImpleFix 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 4.1.2.2) as described in Section 4.1.2.3.

4.1.2.1.1 Control dependence. ImpleFix 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 Figure 2.1.

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, $\ell \not\rightsquigarrow \jmath$ 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 ImpleFix's heuristics: Section 4.1.2.3 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.

4.1.2.1.2 Expression dependence. ImpleFix 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 ImpleFix will be able to affect the value

of any assignable expressions (see Section 4.1.3). 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 $a.\mathit{plus}\ (b)$. 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 $\mathsf{eprox}(i \leq \mathit{count}, 0 \leq i \leq \mathit{count} + 1)$ 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 Figure 2.1, for example, $\mathsf{edep}(\mathit{index}, \mathit{index} = 0)$ is $1/3$ because $\mathsf{eprox}(\mathit{index}, \mathit{index} = 0) = 1$ and $\mathit{index} = 0$ itself has the maximum expression proximity to $\mathit{index} = 0$.

4.1.2.2 Dynamic analysis

Our dynamic analysis borrows techniques from generic fault localization [108] 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 4.1.2.1) to differentiate further among snapshots that receive similar dynamic scores.

4.1.2.2.1 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. [108], 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.

ImpleFix'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 ImpleFix. It is also consistent with recent experimental evidence [95] 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.[4]

4.1.2.2.2 Dynamic score. Assume an arbitrary order on the test cases and let $\sigma(t)$ be α^i for the i-th failing test case t and $\beta\alpha^i$ for the i-th 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).

[4]The results of Wong et al.'s heuristics in Qi et al.'s experiments [95] are not directly applicable to ImpleFix (which uses different algorithms and adapts Wong et al.'s heuristics to its specific needs); replication belongs to future work.

The evidence provided by each test case adds up:

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

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

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

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 ImpleFix 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 4.2 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 ImpleFix, which tend to produce slightly shorter running times on average (up to 10% 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.

4.1.2.3 Overall score

ImpleFix 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 [19]:

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

Our current choice of parameters for the dynamic score (Section 4.1.2.2.2) makes it dominant in determining the overall 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 Figure 2.1, the snapshot $\langle 41, \mathit{idx} > \mathit{index}, \textbf{True} \rangle$ receives a high overall score. ImpleFix targets snapshots such as this in the fix action phase.

4.1.3 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, ImpleFix 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:

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

ImpleFix uses two basic strategies for generating fixing actions: setting and replacement. Setting (described in Section 4.1.3.1) consists of modifying the value of variables or objects through assignments or routine calls. Replacement (described in Section 4.1.3.2) 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 [29, 65], 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 4.1.3.3);

3. For integer expressions, we also deploy constraint solving techniques to build suitable derived expressions (Section 4.1.3.4).

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

4.1.3.1 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 4.1.2.1.2) 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* $>$ **Current**.*index*$\rangle$ in Figure 2.1 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.

4.1.3.1.1 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*.

4.1.3.1.2 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 4.1.3.4).

4.1.3.1.3 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 4.1.3.3 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 2.1.1, the fault's snapshot $\langle 41, \mathit{idx} > \mathit{index}, \textbf{True}\rangle$ determines the settings $\mathsf{set}\langle 41, \mathit{idx} > \mathit{index}\rangle$ that include assignments of 0, 1, and -1 to *idx* and *index*, and unit increments and decrements of the same variables.

4.1.3.2 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 Figure 2.1 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 $\mathit{idx} - 1$ 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.

4.1.3.2.1 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.

4.1.3.2.2 Integer expressions. For e of integer type, 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 4.1.3.4).

Continuing the example of the fault of Section 2.1.1, the snapshot $\langle 41, \mathit{idx} > \mathit{index}, \textbf{True} \rangle$ induces the replacement set replace$\langle 41, \mathit{idx} > \mathit{index} \rangle$ 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).

4.1.3.3 Behavioral models

Some of the fixing actions generated by ImpleFix 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.

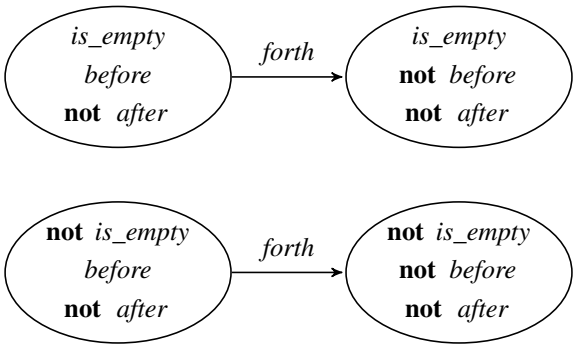


Figure 4.2: Behavioral model of routine *forth*.

Using techniques we previously developed for Pachika [28], ImpleFix 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, Figure 4.2 shows a partial behavioral model for the *forth* routine in Figure 2.1. 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 $\langle 42, \mathit{before}, \textbf{True} \rangle$, 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 4.1.3.1) is impractical.

4.1.3.4 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 Figure 2.1 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, ImpleFix 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* ≥ 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 4.1.3.2) of variable i.

4.1.4 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 4.1.4.1). Then, we *instantiate* the fix schema with the snapshot's predicate p and some fixing action it induces (Section 4.1.4.2).

Whereas the space of all possible fixes generated with this approach is potentially huge, ImpleFix 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 4.2.3.4).

4.1.4.1 Fix schemas

ImpleFix uses a set of predefined templates called *fix schemas*. The four fix schemas currently supported are shown in Figure 4.3;[5] 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.

(*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

Figure 4.3: Fix schemas implemented in ImpleFix.

4.1.4.2 Schema instantiation

For a state snapshot $\langle \ell, p, v \rangle$, we instantiate the schemas in Figure 4.3 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 4.1.3.1) or by replacement ($\mathsf{replace}\langle \ell, p \rangle$ in Section 4.1.3.2).

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*.

[5] Recent work [65] has demonstrated that these simple schemas account for a large fraction of the manually-written fixes found in open-source projects.

For example, consider again the snapshot $\langle 41, \mathit{idx} > \mathit{index}, \textbf{True} \rangle$, which receives a high "suspiciousness" score for the fault described in Section 2.1.1 and which induces, among others, the fix action consisting of decrementing *idx*. The corresponding instantiation of fix schema (b) in Figure 4.3 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 Figure 2.1. Injecting the instantiated schema (replacing line 23) yields the candidate fix in Figure 2.2, already discussed in Section 2.1.1.

4.1.5 Fix Validation

The generation of candidate fixes, described in the previous Sections 4.1.3 and 4.1.4, 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. ImpleFix only reports valid fixes to users, ranked as described in Section 4.1.6.

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 [92, 93], $\mathfrak{S}_r$ 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 [93] 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 4.2), 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 4.2 show that the simple contracts normally available in Eiffel programs are often good enough in many practical cases to enable ImpleFix 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.

4.1.6 Fix Ranking

The ImpleFix 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, ImpleFix 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.[6] Since multiple fixing actions may determine valid fixes for the same snapshot, ties in the ranking are possible. The experiments in Section 4.2 demonstrate that high-quality proper fixes often rank in the top 10 positions among the valid ones; hence ImpleFix users only have to inspect the top fixes to decide with good confidence if any of them is deployable.

4.2 Experimental Evaluation

We performed an extensive experimental evaluation of the behavior and performance of ImpleFix by applying it to over 200 faults found in various Eiffel programs [86]. The experiments characterize the reproducible *average* behavior of ImpleFix 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 [9] to achieve *statistically significant* results in the parts that involve randomization; the faults submitted to ImpleFix 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 Chapter 6). First, the experiments try to capture the usage of ImpleFix as a fully automatic tool where user interaction is limited to selecting a project, pushing a button, and waiting for the results. The second

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

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 4.1.5.

4.2.1 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 ImpleFix correct, and what are their characteristics?

Q2 What is the *quality* of the fixes produced by ImpleFix?

Q3 What is the *cost* of fixing faults with ImpleFix?

Q4 How *robust* is ImpleFix's performance in an "average" run?

The **main findings** of the evaluation are as follows:

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

4.2.2 Experimental Setup

All the experiments ran on the computing facilities of the Swiss National Supercomputing Centre consisting of Transtec Lynx CALLEO High-Performance Servers 2840 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 ImpleFix in a standard programming environment.

We now describe the code bases and the faults targeted by the experiments (Section 4.2.2.1), then present the experimental protocol (Section 4.2.2.2).

4.2.2.1 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 ImpleFix in different conditions.

4.2.2.1.1 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 [76] 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 4.1 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 also offer a better interface with more Boolean queries that can be used by ImpleFix (Section 3.1). The availability of *contracts* also varies significantly in the code bases, ranging from 0.76 precondition clauses

Table 4.1: 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**

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 ImpleFix'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).

4.2.2.1.2 Faults targeted by the experiments. To select a collection of faults for our fixing experiments, we performed a preliminarily run of AutoTest [70] 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 3.4). Table 4.2 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 ($\frac{\#\mathrm{F}}{\mathrm{kLOC}}$). 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:

Table 4.2: 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**

- 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 ImpleFix: 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 [93], 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.

4.2.2.2 Experimental protocol

The ultimate goal of the experiments is to determine the typical behavior of ImpleFix 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 ImpleFix sessions, to minimize the dependency of the quality of fixes produced by ImpleFix on the choice of test cases, and to avoid requiring users to provide test cases.

For each unique fault f identified as in Section 4.2.2.1, 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 ImpleFix algorithms are deterministic, an m-minute series on some fault f uniquely determines an ImpleFix 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 ImpleFix 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).

4.2.3 Experimental Results

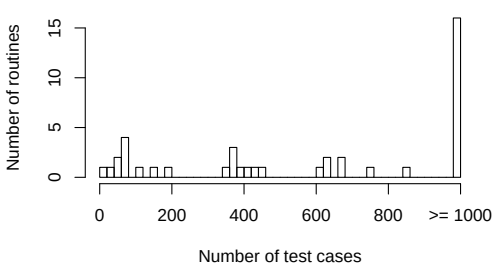
The experimental data were analyzed through statistical techniques. Section 4.2.3.1 examines the code bases from testing point of view. Section 4.2.3.2 discusses how many valid fixes ImpleFix produced in the experiments, and Section 4.2.3.3 how many of these were proper fixes. Section 4.2.3.4 presents the average ImpleFix running times. Section 4.2.3.5 analyzes the performance of ImpleFix over multiple sessions to assess its average behavior and its robustness.

4.2.3.1 Testability of the experimental subjects

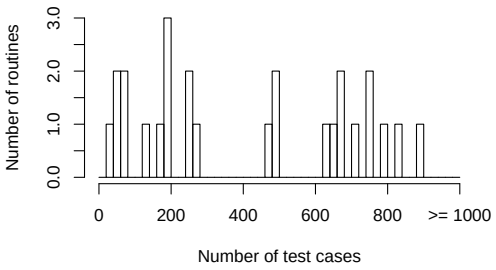
For the evaluation, what matters most is the number and quality of fixed produced by ImpleFix. It is interesting, however, to look into the results of AutoTest sessions to get a more precise characterization of the experimental subjects and to see how the four code bases differ in their testability. The data provides more evidence that the four code bases have different quality and are diverse subjects for our experiments.

4.2.3.1.1 Total number of tests. Each histogram in Figure 4.4 depicts the distribution of the mean number of test cases generated by AutoTest in the 30 repeated 60-minute sessions for each routine. That is, a bar at position x reaching height y denotes that there exist y routines $r_1, \ldots, r_y$ such that, for each $1 \leq j \leq y$, the mean number $|T|$ of tests T in the 60-minute series on some fault of r_j is x. Figures 4.4a–d show the distributions of the individual code bases, while Figure 4.4e is the overall distribution.

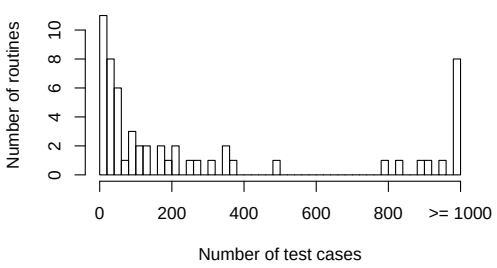
The figures suggest that `Base` is normally easily testable—probably a consequence of its carefully-designed interface and contracts. In contrast, `Cards` and



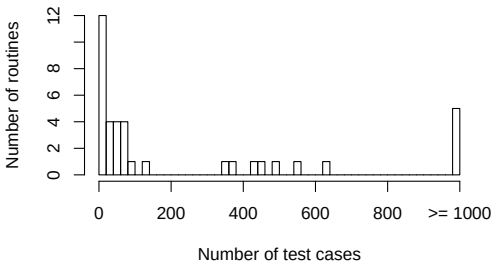
(a) `Base`



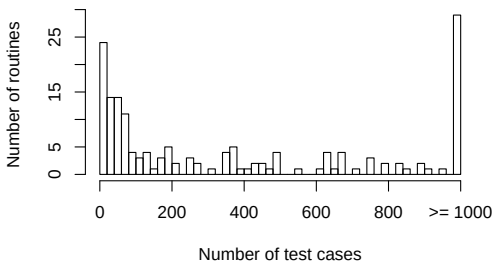
(b) `TxtLib`



(c) `Cards`



(d) `ELearn`



(e) All code bases

Figure 4.4: Number of tests generated by AutoTest on the experimental subjects.

`ELearn` are hard to test on average; and `TxtLib` is a mixed case. A Mann-Whitney U test confirms that the differences are statistically significant: if we partition the four code bases into two groups, one comprising `Base` and `TxtLib` and the other `Cards` and `ELearn`, intra-group differences are not statistically significant (with $692 \leq U \leq 1272$ and $p > 0.06$) whereas inter-group differences are (with $264 \leq U \leq 1754$ and $p < 0.03$).[7]

4.2.3.1.2 Ratio of failing to passing tests. Another interesting measure is the average ratio of failing to passing tests generated in one session, which gives an idea of how frequent failures are. Each histogram in Figure 4.5 depicts the distribution of the mean failing-to-passing ratio for the test cases generated by AutoTest in the 30 repeated 60-minute sessions for each routine; notice that the horizontal scale is logarithmic. That is, a bar at position x reaching height y denotes that there exist y routines $r_1, \ldots, r_y$ such that, for each $1 \leq j \leq y$, the mean ratio $|F|/|P|$ of failing tests F to passing tests P in the 60-minute series on some fault of r_j is e^x.

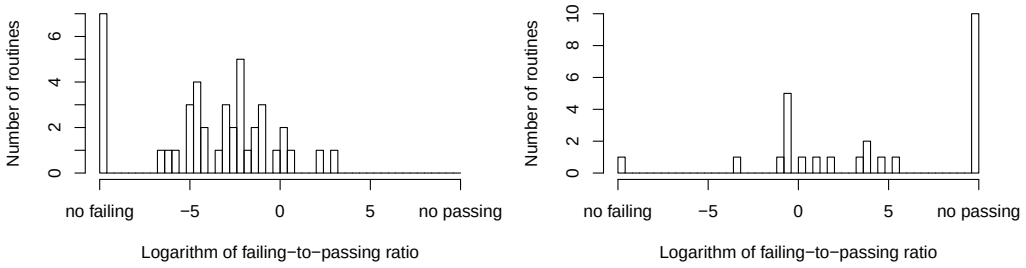
Consistently with Figure 4.4, Figure 4.5 suggests that it is harder to produce failing tests for `Base` than for the other code bases. A Mann-Whitney U test confirms that the difference between `Base` and the other three code bases is statistically significant (with $105 \leq U \leq 445$ and $p < 10^{-7}$) whereas the differences among `TxtLib`, `Cards`, and `ELearn` are not (with $515 \leq U \leq 859$ and $p > 0.06$).

4.2.3.2 How many faults ImpleFix can fix

It is important to know for how many faults ImpleFix managed to construct *valid* fixes in *some* of the repeated experiments. The related questions of whether these results are sensitive to the testing time or depend on chance are discussed in the following sections.

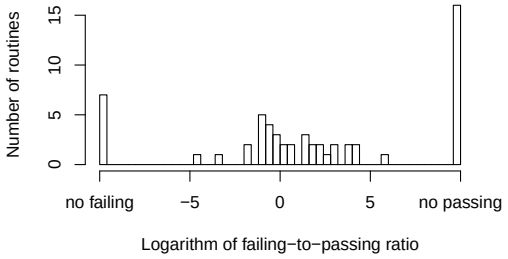
4.2.3.2.1 When ImpleFix succeeds. The second column of Table 4.3 lists the total number of unique faults for which ImpleFix was able to build a *valid* fix and *rank* it among the top 10 during *at least one* of the 55080 (270 sessions for each of the 204 unique faults) fixing sessions, and the percentage 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 4.2. Overall, ImpleFix succeeded in fixing 86 (or 42%) of the faults. Section 4.2.3.5 discusses related measures of *success rate*, that is the percentage of sessions that produced a valid fix.

[7] In this section, the sample sizes for the U tests are the number of faults in each code base.

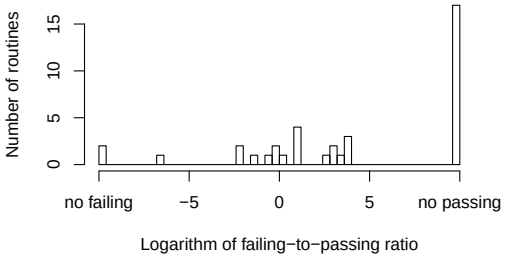


(a) `Base`

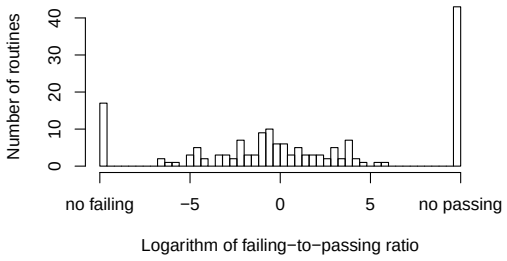
(b) `TxtLib`



(c) `Cards`



(d) `ELearn`



(e) All code bases

Figure 4.5: Failing-to-passing ratio of tests generated by AutoTest on the experimental subjects; the horizontal scales are logarithmic.

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.

4.2.3.2.2 When ImpleFix fails. To understand the limitations of our technique, we manually analyzed all the faults for which ImpleFix always failed, and identified four scenarios that prevent success. Table 4.4 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 4.2.2.1) could not be reproduced during the shorter AutoTest sessions used to provide input to ImpleFix (Section 4.2.2.2). Without failing test cases[8] the ImpleFix algorithms cannot possibly be expected to work. Column #NoFail in Table 4.4 lists the faults that we could not reproduce, and hence could not fix, in the experiments.[9]

Complex patches required. While a significant fraction of fixes are simple [29], some faults require complex changes to the implementation (for example, adding a loop or handling special cases differently). Such patches are currently

[8] As a side remark, ImpleFix managed to fix 19 faults for which AutoTest could generate *only failing* tests; 7 of those fixes are even proper.

[9] Even if AutoTest were given enough time to generate failing tests, ImpleFix would still not succeed on these faults due to complex patch required (4 faults) or incorrect contracts (6 faults).

Table 4.3: Number of faults fixed by ImpleFix (*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 4.4: Types of faults that ImpleFix 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**

out of the scope of ImpleFix; column #Complex of Table 4.4 lists the faults that would require complex patches.

Incorrect or incomplete contracts. ImpleFix assumes contracts are correct and tries to fix implementations based on them. In practice, however, contracts contain errors too; in such cases, ImpleFix 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 ImpleFix's attempts thus become vain. Column #Contract of Table 4.4 lists the faults involving incorrect or incomplete contracts that ImpleFix cannot fix. (In recent work [85], 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 ImpleFix can do. The design flaws that ImpleFix 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 4.4 lists the faults due to design flaws that ImpleFix cannot fix.

4.2.3.2.3 Which fix schemas are used. Not all four schemas available to ImpleFix (Section 4.1.4.1) are as successful at generating valid fixes. Table 4.5 shows the number of faults successfully fixed using each of the schemas *a*, *b*, *c*, and *d* in Figure 4.3. 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. Schemas *b* and *d* are the most successful ones, producing valid fixes for 79% and 75% 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*).

Table 4.5: Number of faults fixed using each of the fix schemas in Figure 4.3.

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**

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

4.2.3.3 Quality of fixes

What is the quality of the valid fixes produced by ImpleFix 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 4.1.5).

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.

Table 4.6: Number of faults fixed by ImpleFix (*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 4.7: Number of faults with proper fixes using each of the fix schemas in Figure 4.3.

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**

The second column of Table 4.6 lists the total number of unique faults for which ImpleFix 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 4.2 and 4.3. Overall, ImpleFix produces proper fixes in the majority (59% of 86 faults) of cases where it succeeds, corresponding to 25% of all unique faults considered in the experiments; these figures suggest that the quality of fixes produced by ImpleFix 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 6.5.

We did not analyze the ranking of proper fixes within the top 10 valid fixes reported by ImpleFix. The ranking criteria (Section 4.1.6) are currently not precise enough to guarantee that proper fixes consistently rank higher than improper ones. Even if the schemas used by ImpleFix lead to textually simple fixes, analyzing up to 10 fixes may introduce a significant overhead; nonetheless, especially for programmers familiar with the code bases[10], 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 ImpleFix.

4.2.3.3.1 Which fix schemas are used. The effectiveness of the various fix schemas becomes less evenly distributed when we look at proper fixes. Table 4.7 shows the number of faults with a proper fix using each of the schemas *a*, *b*, *c*, and *d* in Figure 4.3; it is the counterpart of Table 4.5 for proper fixes. schema *a*

[10]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.

is used in no proper fix, whereas schema *b* is successful with 78% of the 51 faults for which ImpleFix 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 [82] 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 ImpleFix's fault localization based on Boolean conditions in snapshots naturally leads to conditional fixes).

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

4.2.3.4 Time cost of fixing

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

4.2.3.4.1 Fixing time per fault. Figure 4.6 shows the distribution of running times for ImpleFix (independent of the length of the preliminary AutoTest sessions) in all the experiments.[11] 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 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 11670 "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 ImpleFix requires limited amounts of time in general.

Table 4.8 presents the same data about non-empty fixing sessions in a different form: for each amount of ImpleFix 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 4.9 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.

[11] ImpleFix ran with a timeout of 60 minutes, which was reached only for two faults.

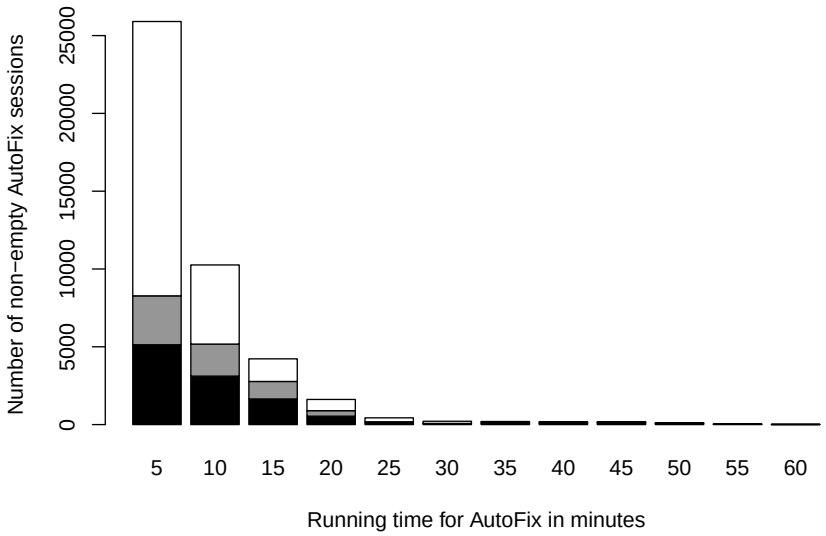


Figure 4.6: Distribution of running times for ImpleFix, independent of the length of the preliminary AutoTest sessions (black bars: sessions with proper fixes; gray bars: sessions with valid fixes; white bars: all sessions).

Table 4.8: Distribution of running times for ImpleFix.

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 4.9: ImpleFix 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

4.2.3.4.2 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 ten (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 ImpleFix for t_2 and produces $v > 0$ valid fixes as $(t_1 + t_2)/v$. Figure 4.7 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 4.10 shows the statistics of unit fixing times for all sessions producing valid fixes, and for all sessions producing proper fixes. Figure 4.8 shows the same distribution of unit fixing times as Figure 4.7 but for proper fixes. This distribution is also skewed towards shorted fixing times, but much less so than the one in Figure 4.7: 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.

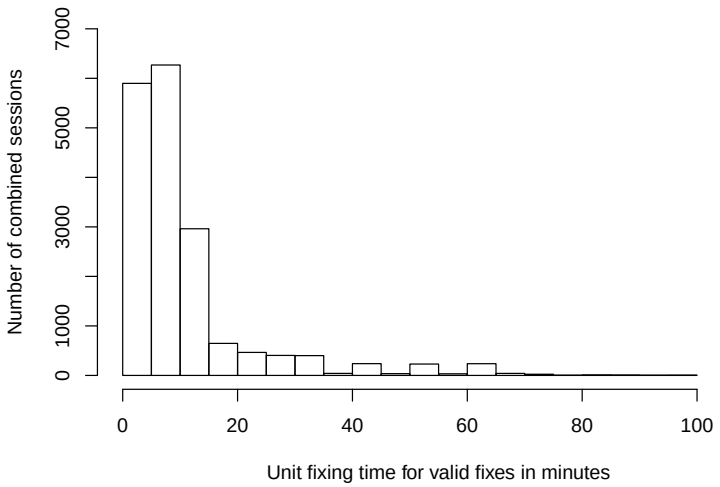


Figure 4.7: Distribution of unit fixing times for *valid* fixes (which including the time spent in the preliminary AutoTest sessions).

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

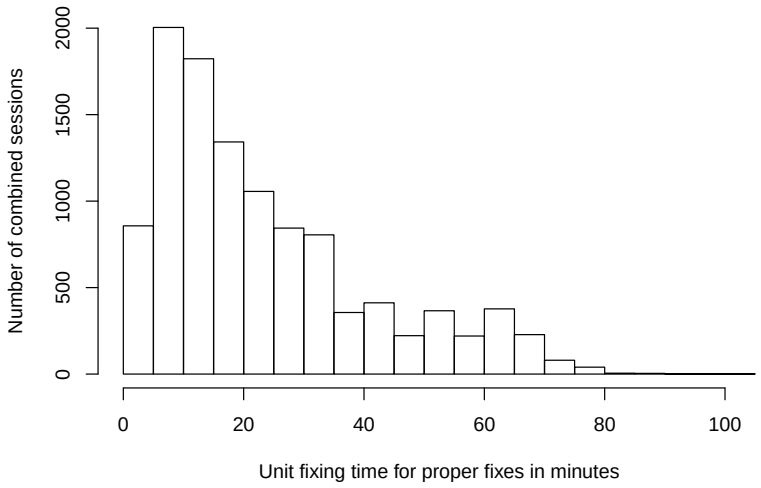


Figure 4.8: Distribution of unit fixing times for *proper* fixes (including the time spent in the preliminary AutoTest sessions).

Table 4.10: 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

divided by the total number of valid fixes produced (assuming we get at least one valid fix). Table 4.11 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 ImpleFix (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 ImpleFix and AutoTest: as a rule of thumb, running AutoTest for five to ten minutes has a fair chance of producing test cases for ImpleFix to correct an “average” fault.

Table 4.11: 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**

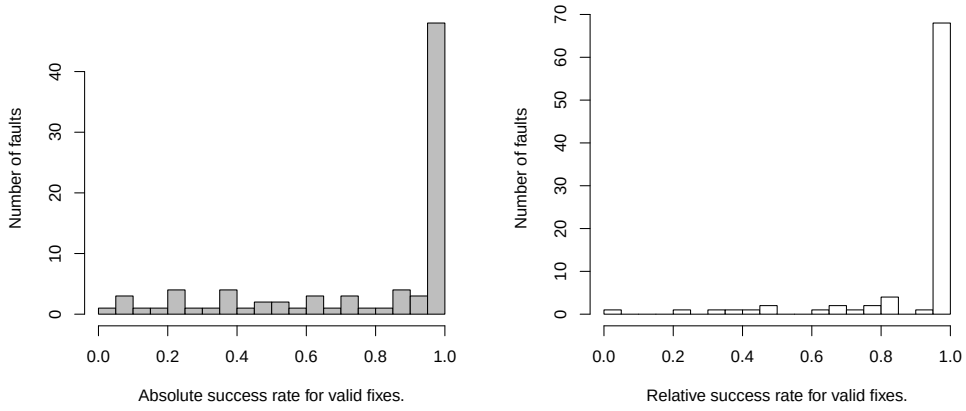
In our experiments, ImpleFix took on average less than 20 minutes per valid fix, including the time required to generate suitable tests with AutoTest.

4.2.3.5 Robustness

The last part of the evaluation analyzes the robustness and repeatability of ImpleFix sessions. The ImpleFix 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 ImpleFix with respect to different choices of input test suites, that is how ImpleFix's output depends on the test cases supplied. Assessing robustness is important to demonstrate that our evaluation is indicative of *average* usage, and its results do not hinge on having used a particularly fortunate selection of tests.

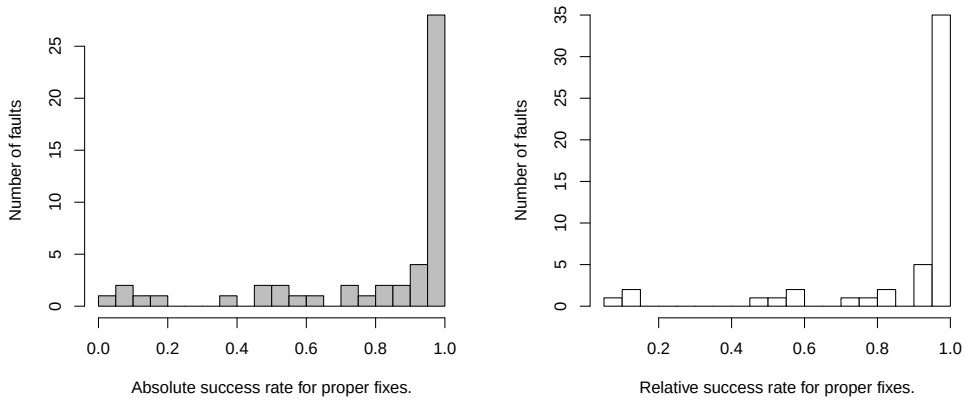
Our experiments consisted of many repeated runs of AutoTest, each followed by ImpleFix runs using the generated test as input. To assess robustness we fix the testing time, and we measure the percentage of ImpleFix runs, on each of the repeated testing sessions terminating within the allotted testing time, that produced a valid fix. A high percentage shows that ImpleFix 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 $\mathfrak{f}$ and a testing time m, the



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

Figure 4.9: Distribution of success rates for *valid* fixes.



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

Figure 4.10: Distribution of success rates for *proper* fixes.

m-minute absolute success rate on f is defined as the percentage of m-minute fixing sessions on 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). Figure 4.9 shows the distribution of the absolute (Figure 4.9a) and relative (Figure 4.9b) success rates for all "fixable" faults—for which ImpleFix produced a valid fix at least once in our experiments—for any testing time m. The graphs demonstrate that ImpleFix 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 4.2.3.2) and thus characterize the robustness of ImpleFix's behavior on the "approachable" faults. (The fact that a classification into "approachable" and "hard" faults for ImpleFix 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 4.12 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 $X = 50, 80, 90, 95$.[12] Each table entry also shows, in parentheses, the percentage of the fixed faults, relative to the 86 fixable faults that ImpleFix fixed at least once; the data is shown for both the relative and the absolute success rate. For example, ImpleFix was successful at least 95% of the times with 56% of all fixable faults; or even with 79% 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% of all sessions and 41% of all non-empty sessions produced a valid fix. These success rates suggest a high repeatability of fixing.

Figure 4.10 and Table 4.13 display similar data about successful sessions that produced at least one *proper* fix, with percentages relative to all faults for which ImpleFix produced a proper fix at least once in our experiments. The aggregated data over all fixing sessions for all faults is the following: 20% of all sessions and 25% of all non-empty sessions produced a proper fix; these percentages are quite close to the 25% of all faults for which ImpleFix produces at least once a proper fix (Table 4.6). 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 ImpleFix's behavior

[12] 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 4.12: Repeatability of ImpleFix 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%)**

Table 4.13: Repeatability of ImpleFix 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%)**

is often robust and largely independent of the quality of provided test cases.

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

4.2.4 Limitations

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

Contracts or a similar form of annotation must be available in the source code. The simple contracts that programmers write [38] are sufficient for ImpleFix; 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 [38], which restricts broader applicability of ImpleFix 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 [73]) and are comparable to those required by other more accepted practices. For example, EiffelBase's contracts-to-code ratio is around 0.2 [93]; 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 [13, 74, 66] 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 ImpleFix, 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 4.2.2.1). Nonetheless, the expressiveness of contracts is significant, and in fact we could identify various categories of contract-violation faults that ImpleFix can or cannot fix (Section 4.2.3.2).

Correctness of contracts is assumed by AutoTest, which uses them as oracles, and by ImpleFix, which fixes implementations accordingly. Since contracts have errors too, this may affect the behavior of ImpleFix on certain faults (see Section 4.2.3.2). 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 ImpleFix include only a subset of all possible actions (Section 4.1.3) and are limited to simple schema (Section 4.1.4). This limits the range of fixes that ImpleFix 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 [29, 65].

4.2.5 Threats to validity

While we designed the evaluation of ImpleFix 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 [13] 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 ImpleFix 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 exhibiting variety in terms of quality, maturity, and available contracts—within the constraints imposed by our fundamental design choices, as discussed in Section 4.2.2.1.

Variability of performance relative to different choices for the various heuristics used by ImpleFix 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 4.1.2.2) whose impact we have not investigated as thoroughly as other aspects of the ImpleFix algorithm. As also discussed in Section 4.1.2.2, 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 [54]. Our motivation for this choice is that we conceived ImpleFix 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, ImpleFix 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 [53].

Programmer-written contracts were used in all our experiments. This ensures that ImpleFix 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 ImpleFix performance is affected. In recent work [93]

we obtained good results with this approach applied to testing with AutoTest.

CHAPTER 5

CORRECTING THE SPECIFICATION

Using the ImpleFix technique from the Chapter 5 AutoFix can already generate high quality fixes for many program faults. There are, however, a significant number of bugs [20] whose most appropriate correction is changing the specification to rectify the expectations about what the implementation ought to do. For example, a function *max* computing the maximum value of a set of integers is undefined if the set is empty; we could change *max*'s implementation to return a special value when called on an empty set, but the best thing to do is disallowing such calls altogether by specifying them invalid.

This chapter presents the SpeciFix technique that automatically fixes bugs by correcting the specification [85]: given a program execution that violates some contract, and therefore reveals a bug, the technique suggests changes to the contracts that prevent the violation from being triggered.

Fixing contracts relies on extracting specification elements based on the actual behavior of the implementation. This is superficially similar to the problem of *inferring* (or mining) specifications—a well-established research area that produced numerous landmark results (e.g., [24, 36]; see Section 6.6 for more references). While SpeciFix uses inference techniques as one of its components, suggesting changes to an existing specification to correct a bug is more delicate business than just inferring specifications. Changing contracts is changing the design of an API as experienced by its clients. In the example of *max*, adding a precondition that requires that the set be non empty makes all client code of *max* responsible for satisfying the requirement upon calling *max*. Therefore, we must make sure that the suggested contract changes have a limited impact on a potentially infinite number of clients.

The SpeciFix technique presented in this section uses a combination of heuristics to validate possible specification fixes with respect to their impact on client code. It discards fixes that invalidate previously passing test cases; to avoid over-

fitting, it runs every candidate fix through a regression testing session that generates (completely automatically, using our testing framework AutoTest) new executions; and it ranks all fixes that pass regression by preferring those that are the least restrictive. The empirical evaluation in Section 5.2 indicates that these heuristics work well in practice for the bugs we considered. Notably, there is a significant fraction of bugs whose appropriate fix is a change to the specification; in those cases, SpeciFix can often generate useful fixes.

Section 5.1 presents the technique implemented in SpeciFix, starting with an overview of its components (Figure 5.1) followed by a detailed description of each of them. The evaluation in Section 5.2 presents experiments where we applied SpeciFix to 44 faults in standard data-structure libraries.

5.1 How SpeciFix Generates Corrections to Contracts

Just like generating fixes to the implementation, SpeciFix works completely automatically when generating corrections to contracts: its only input is an Eiffel program annotated with simple contracts. After going through the steps described in the rest of this section, SpeciFix's final output is a list of fix suggestions to contracts for the bugs in the input program.

Figure 5.1 gives an overview of the components of the SpeciFix technique for fixing the contracts. The technique is also based on dynamic analysis, and hence it characterizes correct and incorrect behavior by means of passing and failing *test cases* (Sections 3.4), e.g. the ones produced using AutoTest. The core of the *fix generation* algorithm applies two complementary strategies (Section 5.1.1): weaken (i.e., relax) a violated contract if it is needlessly restrictive; or strengthen an existing contract to rule out failure-inducing inputs. SpeciFix produces *candidate fixes* using both strategies, possibly in combination (Section 5.1.2). To determine whether the weaker or stronger contracts remove all faulty behavior in the program, SpeciFix runs candidate fixes through a *validation* phase (Section 5.1.3) based on all available tests. To avoid overfitting, some tests are generated initially but used only in the validation phase (and not directly to generate fixes). If multiple fixes for the same fault survive the validation phase, SpeciFix outputs them to the user *ordered* according to the strength of their new contracts: weaker contracts are more widely applicable, and hence are ranked higher than more restrictive stronger contracts (Section 5.1.3).

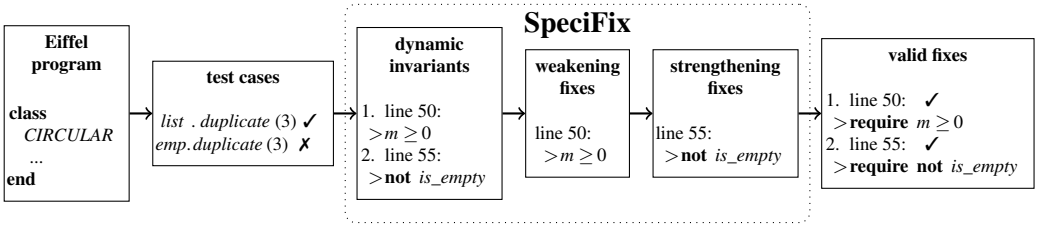


Figure 5.1: How SpeciFix generates fixes to contracts. Running AutoTest on an input Eiffel program with contracts produces a collection of test cases that characterize correct and incorrect behavior. With the goal of correcting faulty behavior, the fix generation algorithm infers dynamic invariants in passing tests (Section 5.1.4) and builds candidate fixes using two strategies: weakening and strengthening the existing contracts (Sections 5.1.1 and 5.1.2). The candidate fixes enter a validation phase where they must pass all valid test cases; valid fixes are ranked—the weaker the new contracts the higher the ranking—and presented as output (Section 5.1.3).

5.1.1 Weakening vs. Strengthening

Let t be a failing test case with call sequence κ_t as in (3.2); $r = r_0$ is the outermost routine of t, and r_n is the routine whose contract violation triggers the fault. Assuming the implementation of all routines $r_0, \ldots, r_n$ is correct, we should change the contracts of $r_0, r_1, \ldots, r_n$ to fix the fault exposed by t. There are two ways to do that:

Strengthening: strengthen r's precondition to disallow t's input. Strengthening makes t invalid and thus prevents the call sequence that led to the violation of r_n's contract.

Weakening: weaken r_n's contract to allow t's execution to continue past r_n. If the execution can continue without triggering other errors, weakening makes t passing.

If applicable, weakening is in principle preferable to strengthening, because the former does not risk breaking clients by introducing more stringent conditions for correctly calling r. Strengthening is, however, always applicable, whereas weakening may not work if r_n's correct execution depends on the weakened contract. Even in the cases where weakening makes t passing without triggering any new fault, it may be that the absence of new faults is just a result of the rest of the specification being inaccurate or incomplete. For example, weakening the precondition of a function *max* to work on lists of any size (including empty lists) may not trigger any faults simply because *max* has no postcondition, and hence there is no automatic way of finding out that the value returned for empty lists is inconsistent.

In practice, SpeciFix prefers the least restrictive fixes (i.e., weakening) but always tries both weakening and strengthening in combination. Another observation is that strengthening only the outermost routine's precondition often is too *ad hoc*, since it corresponds to a partial change of API assumptions which may be inconsistent with the way other routines are used. Therefore, SpeciFix tries to collectively strengthen all routines $r_0, \ldots, r_{n-1}$ to disallow fault-inducing input at every call site. Indeed, the experiments of Section 5.2 show that strengthening leads to many useful and correct fixes in practice.

5.1.2 Fix Generation

A run of SpeciFix targets a specific fault of some routine r. This is characterized by a set $\mathcal{F}_r$ of failing test cases all of which have r as outermost routine and identify the same fault—the violation of contract A_n (pre- or postcondition) of routine r_n. To characterize correct behavior, SpeciFix also inputs a set $\mathcal{P}_r$ of passing test cases which have r as outermost routine. Based on this, SpeciFix builds a set Φ of candidate fixes through the following steps, illustrated on the running example.

Build weakening assertions Ω for r_n. Let $\widetilde{r}_n$ be r_n with A_n relaxed to **True**. Generate fresh sets $\widetilde{\mathcal{P}}$ and $\widetilde{\mathcal{F}}$ of passing and failing test cases for $\widetilde{r}_n$. Based on them, determine the sets $\mathcal{I}^{\widetilde{\mathcal{P}}}$ and $\mathcal{I}^{\widetilde{\mathcal{F}}}$ of dynamic *invariants* respectively holding in all passing tests $\widetilde{\mathcal{P}}$ and in all failing tests $\widetilde{\mathcal{F}}$ (Section 5.1.4 describes the dynamic invariant detection process). Let $\Omega = \{\omega \mid \omega \in \mathcal{I}^{\widetilde{\mathcal{P}}} \text{ and } \neg\omega \in \mathcal{I}^{\widetilde{\mathcal{F}}}\}$ be a set of weakening assertions, which characterize the minimal requirements for a test of $\widetilde{r}_n$ to be passing and not failing. In the example, *make* works without errors when $m \geq 0$, whereas it fails when $m < 0$; thus $\Omega = \{m \geq 0\}$.

Build weakening fixes W. For each $\omega \in \Omega \cup \{\textbf{False}\}$, build the weakening fix f obtained by replacing A_n with $A_n \vee w$ in r_n. Add f to the set W of weakening fixes. Adding **False** to Ω determines a dummy fix which is used to build purely strengthening fixes in the next step. In the example, W contains a weakening fix f_w corresponding to the one in Figure 2.4c, and a dummy fix f_0 where *make*'s precondition has been "weakened" with **False** (hence it is unchanged).

Validate weakening fixes. For each $f \in W$, if f passes all tests in $\mathcal{P}_r \cup \mathcal{F}_r$ then add f to the set Φ of candidate fixes without modifications, and remove it from W. In the example, f_w passes validation and is added to Φ. f_0 is instead the unchanged program in Figure 2.3, and hence it stays in W.

Build strengthening assertions Σ_k for r_k. For each $f \in W$ that did not pass validation, determine the sets $\mathcal{I}_k^{\mathcal{P}}$ and $\mathcal{I}_k^{\mathcal{F}}$ of dynamic *invariants* currently holding in all pre-states of the calls to r_k respectively in the passing tests $\mathcal{P}_r$ and in the failing tests $\mathcal{F}_r$; k ranges over the subset of $\{0, \ldots, n-1\}$ for which s_k

is a pre-state ($s_k r_k$ appears in the traces). Let $\Sigma_k = \{\sigma \mid \sigma \in \mathcal{I}_k^{\mathcal{P}} \text{ and } \neg\sigma \in \mathcal{I}_k^{\mathcal{F}}\}$ be the corresponding sets of strengthening assertions, which characterize the minimal additional requirements for a test to pass through r_k without failing. In the example, *duplicate* correctly calls *make* precisely when *count* > 0; thus, $\Sigma_0 = \{\mathit{count} > 0\}$.

Build strengthening fixes. For each combination $\langle \sigma_0, \ldots, \sigma_{n-1} \rangle \subseteq \Sigma_0 \times \cdots \times \Sigma_{n-1}$ of strengthening assertions, build the strengthening fix ϕ obtained by replacing each precondition P_{r_k} of routine r_k with $P_{r_k} \wedge \sigma_k$, for all applicable k. Add ϕ to the set Φ of candidate fixes. In the example, the dummy fix f_0 is turned into a valid fix ϕ_0 by strengthening *duplicate*'s precondition as *count* > 0.

Candidates. The output of the fix generation phase is a set Φ of fix candidates. The candidates are filtered and ranked as explained in the following section.

5.1.3 Fix Validation and Ranking

Validation. The purpose of the *validation* phase is to ascertain which of the candidate fixes in Φ remove the fault under analysis. To this end, SpeciFix runs every fix candidate $f \in \Phi$ through all available tests for r; f is *valid* if it still passes all originally passing tests, and it also passes all originally failing tests that have not become invalid.

The dual risk of unsoundness for validation based on a finite number of test cases is *overfitting*: a fix may pass validation but be unusable in a general context, because it introduces specification changes that harm usages of the API different from those exercised by the test cases used to generate the fix. To reduce the risk of overfitting, SpeciFix uses only half of the originally generated test cases to generate the candidate fixes. Then, the validation phase uses *all* available tests for the routine under analysis, not only those in $\mathcal{P}_r$ and $\mathcal{F}_r$ used to generate fixes. This increases the likelihood that the validated fixes are applicable beyond the specific cases that drove fix generation.

Ranking. Not all valid fixes are equally desirable: all else being equal, we prefer those that introduce the least changes to the specification, and that make invalid the fewest test cases. SpeciFix ranks valid fixes to reflect these criteria, and only reports the top five fixes for each fault. This approach is a good compromise between the contrasting needs of exposing programmers to a limited number of fixes—which they have to understand and validate—and of retaining fixes that fall behind in the ranking even if they are of high quality, due to the imperfect precision of the ranking heuristics.

The ranking heuristics is based on two elements: number of invalidated tests and the strength of the new contracts. A fix f consists of a collection $\langle A_0, \ldots, A_n \rangle$ of new contracts for the routines $r_0, \ldots, r_n$; each A_k $(0 \leq k \leq n)$ is either a pre-

or a postcondition and may be weaker, stronger, or unchanged with respect to the original program. Given two valid fixes f_1, f_2, let A_k^1, A_k^2 be their new contracts for the same routine r_k. We say that A_k^1 is *not stronger than* A_k^2, written $A_k^1 \preceq A_k^2$, if A_k^1 holds whenever A_k^2 holds; precisely, we determine strength based on executing all available tests for r: $A_k^1 \preceq A_k^2$ iff every test that is valid for A_k^1 (i.e., a test that leads to executions where A_k^1 is evaluated and holds) is also valid for A_k^2 (i.e., A_k^2 is evaluated and holds). This generalizes to an ordering between fixes by lexicographic generalization of $\preceq$ on tuples $\langle A_0, \ldots, A_n \rangle$. The ordering is partial because the sets of valid test cases for f_1 and for f_2 may be non-comparable. The final ranking orders fixes according to the $\preceq$ relation and, for incomparable fixes, ranks higher those that determine the higher number of valid (and hence passing) tests.

In the running example, the weakening fix in Figure 2.4c ranks higher than the strengthening fix in Figure 2.4b: all test cases with *count* >0 are equivalent for the two fixes, but the test cases with *count* = 0 are valid only for the weakening fix.

5.1.4 Dynamic Invariants and State Abstraction

SpeciFix infers invariants at program states dynamically by observing the behavior during concrete executions. Dynamic invariant inference (see Section 6.6) has become a standard technique of dynamic analysis. Using the notation of Section 3.2, we can define an invariant at the entry of routine r_k as an assertion I such that $s_k \models I$ for every passing test t whose trace ρ_t includes the snapshot $s_k\, l_{\overline{r_k}}$; the invariant at routine exit is defined similarly with respect to post-states.

Invariant inference in SpeciFix must cater to the specific needs of fixing contracts. To this end, we abstract the concrete program state by a number of predicates that include public queries and any subexpressions of the available contracts (see Section 3.1).

5.2 Experimental Evaluation

We performed a preliminary evaluation of the behavior of SpeciFix by applying it to 44 bugs of production software. The overall goal of the evaluation is corroborating the expectation that, for bugs whose "most appropriate" correction is fixing the specification, SpeciFix can produce repair suggestions of good quality. A more detailed evaluation taking into account aspects such as robustness and readability of the produced fixes belongs to future work.

CLASS	LOC	#R	#P	#Q	#C	#F	$\#\mathcal{P}$	$\#\mathcal{F}$	$\mathcal{T}_t$	$\mathcal{T}_f$
ACTIVE_LIST	2165	139	91	121	25	2	212	210	240	23
ARRAY	1474	101	70	110	10	9	850	555	900	72
ARRAYED_CIRCULAR[1]	1907	133	80	92	23	3	320	234	360	17
ARRAYED_SET	2346	146	118	131	26	6	554	432	720	34
DS_ARRAYED_LIST	2862	168	219	173	15	3	132	89	240	15
DS_HASH_SET	3159	171	154	140	20	1	14	60	120	5
DS_LINKED_LIST	3497	162	207	166	13	3	360	25	360	25
LINKED_LIST	1995	109	70	91	23	0	–	–	60	–
LINKED_SET	2347	122	99	101	26	4	416	70	480	22
TWO_WAY_SORTED_SET	2856	141	118	118	31	13	1260	655	1260	106
TOTAL	24608	1392	1226	1243	212	44	4118	2330	4680	319

Table 5.1: Classes used in the experiments; for each class we report: lines of code LOC, number #R of routines, number #P of assertions in preconditions, number #Q of assertions in postconditions, and number #C of assertions in the class invariant. In the right-hand side, we report the number #F of faults targeted by the experiments, the total number of test cases (passing $\#\mathcal{P}$ and $\#\mathcal{F}$ failing) used by SpeciFix, the $\mathcal{T}_t$ minutes spent running AutoTest on routines of the class, and the $\mathcal{T}_f$ minutes spent running SpeciFix (net of testing time) on faults of the class.

5.2.1 Experimental Setup

We selected 10 of the most widely used data-structure classes of the EiffelBase (rev. 92914) and Gobo (rev. 91005) libraries—the two major Eiffel standard libraries. While these are the same classes used in the experimental evaluation of ImpleFix (Section 4.2), we did not attempt a direct comparison for different reasons. First, some of the bugs used in ImpleFix have been fixed in the latest library versions, and hence they are not reproducible. Second, ImpleFix and SpeciFix are complementary approaches: our experience with ImpleFix suggested that there is a substantial fraction of bugs whose most appropriate correction is fixing the specification, and it is precisely on those that we expect SpeciFix to work successfully. Third, running SpeciFix on the very same input as ImpleFix would limit the generalizability of the evaluation results; instead, we want to evaluate the behavior of SpeciFix in standard conditions and avoid overfitting.

All the experiments ran on a Windows 7 machine with a 2.6 GHz Intel 4-core CPU and 16 GB of memory. We ran AutoTest for one hour on each of the 10 classes in Table 5.1. This automatic testing session found 44 unique faults consisting of pre- or postcondition violations. We ran SpeciFix on each of these faults individually, using only half of the test cases (randomly picked among those generated for each fault in the one-hour session) to generate the fixes and all of

[1] Shortened to *CIRCULAR* in Section 2.1.2.

them in the validation phase (Section 5.1.3). The right-hand side of Table 5.1 reports, for each class, the total number of test cases used by SpeciFix, and the total time for testing (the initial one-hour sessions plus additional calls to AutoTest to generate tests for relaxed routines used to infer the weakening assertions Ω, as described in Section 5.1.2) and fixing. The average figures *per fault* are: 106.4 minutes of testing time and 7.2 minutes of fixing time (minimum: 4.1 minutes, maximum: 30 minutes, median 6.2 minutes). The testing time dominates since AutoTest operates randomly and thus generates many test cases that will not be used (such as passing tests of routines without faults).

5.2.2 Results

Evaluating the effectiveness of repairs that modify contracts is a somewhat subtle issue, since it ultimately involves what is a design choice: changing API specification. Related work on automatic repair (see Section 6.3) has rarely, if ever,[2] assessed the quality and *acceptability* for human programmers of the produced fixes beyond running standard regression test suites. To this end, in previous work [103, 87] we introduced the notions of *valid* and *proper* fix: any fix that passes all the available tests is valid (and hence every fix output by SpeciFix is valid), but only those that manual inspection reveals to satisfactorily remove the real source of failure without introducing other bugs are classified as *proper*. Even if the line between proper and improper might be fuzzy in some corner cases, we could normally confidently classify fixes into proper and improper based on our familiarity with the code base under analysis.

We use the same classification criterion in the evaluation of fixes produced by SpeciFix: Table 5.2 lists the total number of faults for which SpeciFix generated valid or proper fixes (and ranked them in the top 5 positions: we ignore fixes that rank lower).

For 25% of the faults, SpeciFix produced fixes that manual inspection revealed to satisfactorily remove the real source of failure.

The percentage of proper fixes (25% of faults) is similar to that obtained in the work with SpeciFix; but the high percentage of valid fixes (over 90%) requires some explanation. Obtaining valid contract fixes is easy if only poor-quality tests are available. One can always strengthen preconditions to invalidate failing test cases (or, conversely, weaken failing postconditions to trivially pass tests): since SpeciFix validates fixes based on the available test cases, which in turn are only as good as the contracts of the class (beyond those directly targeted by the fix), such straightforward fixes yield valid repairs for classes equipped with very weak and incomplete contracts. This does not mean that such fixes are always improper;

[2] The only exception we are aware of is [53].

TYPE OF FAULT	#F	VALID	PROPER	VALID FIXES				PROPER FIXES			
				ALL	WEAK	STRONG	BOTH	ALL	WEAK	STRONG	BOTH
Precondition violation	22	22	7	77	23	30	24	13	1	12	0
Postcondition violation	22	20	4	71	56	13	2	7	3	4	0
TOTAL	44	42	11	148	79	43	26	20	4	16	0

Table 5.2: Fixes built by SpeciFix. For each TYPE of fault, the left-hand side of the table reports the number #F of faults of that type input to SpeciFix, and for how many of those faults SpeciFix built (at least one) VALID or PROPER fixes. The right-hand side reports the total number of *fixes* produced in each category; the same fault may have multiple valid or proper fixes. Columns ALL list all fixes in each category, followed by a breakdown into purely weakening (WEAK), purely strengthening (STRONG), and mixed (involving BOTH strengthening of some contract and weakening of some other).

in fact, 80% of all proper fixes strengthen preconditions: it is only when it is combined with very poor specification (especially class invariants) that fixing may lead to improper fixes. Furthermore, despite being not directly deployable, the valid but improper fixes produced by SpeciFix are still very valuable as debugging aids, since they clearly highlight the failure-inducing inputs.

Acceptability trial. In order to get more confidence in the capability of SpeciFix to produce proper, acceptable fixes from a programmer's perspective, we conducted a small trial involving 4 PhD students (henceforth, the "subjects") in our group. The subjects were quite familiar with the Eiffel language and its standard libraries, but had not been involved in the work on ImpleFix or SpeciFix. To keep the workload small, we randomly selected only 8 out of the 11 faults for which SpeciFix produced proper fixes, and submitted them to the subjects: for each fault, we produced one failing test case (randomly picked among those produced by AutoTest) and up to 3 fixes produced by SpeciFix. In order to compare the acceptability of specification and implementation fixes, we also included up to 2 proper implementation fixes for each of 5 faults (out of 8) produced using ImpleFix. For each fault, the subjects: (1) declared which fixes they considered acceptable (i.e., they "correct the fault while not introducing new faults", as in our definition of "proper"); and (2) ordered the fixes in decreasing order of quality.

Table 5.3 summarizes the results of the acceptability trial: for each subject S_x and fault F_y, a "c" represents a contract fix (produced by SpeciFix) and an "i" an implementation fix (produced by ImpleFix); the order represents the one expressed in task (2) of the trial; multiple fixes judged of equally good quality are grouped in braces. Underlined fixes correspond to those judged acceptable in task (1) of the trial. For example, the entry $\{\underline{i}\,\underline{i}\}\,\underline{c}\,c$ in row S_4, column F_5 means that subject S_4 judged the two implementation fixes of fault F_5 acceptable and of

	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8
S_1	$\underline{c}\,\underline{i}$	$\{\underline{c}\,\underline{c}\}\,\underline{i}\,\underline{i}$	$\underline{c}\,\underline{c}\,\underline{i}\,c$	$\underline{i}\,\underline{c}\,\{i\,c\}$	$\{\underline{i}\,\underline{i}\}\,\underline{c}\,c$	$\underline{c}\,\underline{c}$	$\underline{c}\,\underline{c}$	$\underline{c}\,c$
S_2	$\underline{c}\,\underline{i}$	$\underline{c}\,\{c\,i\,i\}$	$\underline{c}\,\underline{c}\,\{i\,c\}$	$\underline{i}\,\underline{c}\,\{i\,c\}$	$\underline{c}\,\underline{i}\,\{c\,i\}$	$\underline{c}\,\underline{c}$	$\underline{c}\,c$	$\underline{c}\,c$
S_3	$\underline{c}\,\underline{i}$	$\underline{c}\,\underline{c}\,\{i\,i\}$	$\underline{c}\,\{c\,i\,c\}$	$\underline{c}\,\underline{i}\,\underline{i}\,c$	$\underline{i}\,\underline{i}\,\{c\,c\}$	$\underline{c}\,c$	$\underline{c}\,c$	$\{c\,c\}$
S_4	$\underline{c}\,\underline{i}$	$\underline{c}\,\underline{c}\,\underline{i}\,\underline{i}$	$\underline{c}\,\underline{i}\,\underline{c}\,c$	$\underline{c}\,\underline{i}\,\{i\,c\}$	$\{\underline{i}\,\underline{i}\}\,\underline{c}\,c$	$\underline{c}\,\underline{c}$	$\{\underline{c}\,\underline{c}\}$	$\underline{c}\,\underline{c}$

Table 5.3: Results of the trial.

equivalent quality, better than one contract fix (still acceptable), and better still than another contract fix (unacceptable).

The disagreement of subject S_3 about which faults are proper targets two faults which are worth discussing in more detail because they are indicative of the expectations of different programmers. Fault F_8 affects routine *subtract* in class *TWO_WAY_SORTED_SET*: *s*. *subtract* (*t*) removes from set *s* all elements in set *t*, but it does not work correctly if *s* and *t* are references to the same object. One repair produced by SpeciFix strengthens *subtract*'s precondition to disallow the case $s = t$; the other three subjects thought that the case of *s* and *t* aliased is special and hence can be handled separately, whereas S_3 maintained that a proper fix should work correctly also when $s = t$. The other fault F_5 affects routine *prune_first* of class *DS_ARRAYED_LIST*: *l*. *prune_first* removes the first element of list *l*, but it does not work if *l* is empty. SpeciFix suggested to strengthen the routine's precondition to *count* > 0; S_3 expected a proper fix to do nothing when *l* is empty, whereas the other subjects thought that removing the "first" element makes sense only if the first element exists. In both cases, how the routines should behave on corner cases is a somewhat subjective matter. Anyway, all the subject agreed that even improper strengthening fixes are useful to understand a fault's source and manually fix them. For F_5 and F_8 which we just described, the inferred preconditions clearly outline the case in which the routines do not work; changing the implementation to cover those cases as well becomes straightforward.

The highlights: all subjects but one agreed with our assessment of proper fixes; the subjects unanimously preferred a contract fix over an implementation fix for 3 of the 5 faults that had both kinds of fix. The subject who disagreed about proper fixes still agreed that the contract fixes for 6 out of 8 faults are proper. With the proviso that its small scale does not warrant arbitrary generalizations, the trial demonstrates substantial agreement with our assessment of proper fixes; and suggests that, if a fault can be fixed with a contract fix, SpeciFix has a chance of building a high-quality one.

Programmers found most proper fixes produced by SpeciFix acceptable and often preferable to fixes for the same bugs that change the implementation.

5.2.3 Limitations and Threats to Validity

Limitations. The main limitation to the applicability of SpeciFix is that it requires contracts, just as ImpleFix does. On the one hand, it requires a language where contracts are expressible; this is an obvious consequence of the technique's goals and is not severely restrictive since many languages support some form of notation for contracts (e.g., JML for Java and CodeContracts for C#). On the other hand, SpeciFix works well only on classes that come already equipped with *some* contracts of decent quality. Class invariants (which SpeciFix does not change but only assumes) are particularly useful to ensure that the test cases generated represent reasonable usage, so that validation (Section 5.1.3) is precise. Despite being often weak and largely incomplete, the kinds of contracts Eiffel programmer write have been sufficient to get good experimental results; but in future work we will investigate how SpeciFix performance improves if it is given more expressive contracts [93].

Threats to validity. The most significant threat to *external* validity—concerning the generalizability of our experimental results—comes from limiting the experiments to data-structure classes. This is a limitation partly inherited from the usage of AutoTest to generate test cases; AutoTest is meant for unit testing and hence works more easily with classes with a clearly defined interface such as data structures. In future work, we plan to experiment with other kinds of program (as we already did successfully with ImpleFix in Section 4.2) and possibly with manually-written test cases. Another threat comes from the small number of subjects used in the trial (Section 5.2.2), and the fact that they all were graduate students. We acknowledge that the trial only gives a preliminary assessment, and more user studies are needed to ensure generalizability.

Threats to *internal* validity—concerning the proper execution of our experiments—include repeatability. Since SpeciFix uses AutoTest to generate test cases, and the performance of AutoTest is affected by chance, different runs may yield different results. Based on our previous extensive experience with using AutoTest's test cases for dynamic analysis as described in Section 4.2, we expect AutoTest behavior to be predictable over the testing time allotted in our experiments; therefore, this threat is unlikely to be significant. Since SpeciFix produces many valid but not proper fixes, an issue is how much effort is required to identify the improper fixes. While we have no hard evidence about this, even improper fixes succinctly characterize the failure-inducing inputs, and hence they are still useful as debugging aids. Furthermore, contract fixes are normally quite simple,

arguably easier to read than implementation fixes; all subjects in the trial spent on average around two minutes to classify each contract fix, which seems to indicate an acceptable overhead. More experiments are also needed to determine the sensitivity of SpeciFix to what fraction of the tests are used for generation vs. validation.

CHAPTER 6

COMPARISON TO EARLIER WORK

We present the related work on automatic fault detection and fixing in six areas: techniques to automatic testing; approaches to fault localization; fixing techniques working on the source code (as AutoFix does); applications to specialized domains; fixing techniques that operate dynamically at runtime; and techniques to inference invariants.

6.1 Automatic testing

During the last decade, automatic testing has become an effective technique to detect faults, in programs and systems, completely automatically. Among the approaches to automatic testing, random testing is one of the simplest, yet it has been successfully applied to a variety of programs including Java libraries and applications [80, 25, 99]; Eiffel libraries [21]; and Haskell programs [22]. The research on random testing has produced a variety of tools—including our own AutoTest [70], Randoop [81], JCrasher [25], Eclat [80], Jtest [83], Jartege [78], Yeti [79], and RUTE-J [6]—as well as rigorous analysis [10] justifying its practical success on theoretical grounds.

Search-based test-case generation refines random testing with the goal of improving its performance and accuracy. McMinn [68] and Ali et al. [3] survey the state of the art in search-based techniques. Genetic algorithms are a recurring choice for searching over unstructured spaces in combination with random exploration; Tonella [100] first suggested the idea, and Andrews et al. [7] show how to use genetic algorithms to optimize the performance of standard random testing. Our previous work [102, 104] also extended purely random testing with search-based techniques. Other approaches to automatic testing introduce white-box techniques such as symbolic execution [62] and fuzzying [41], or leverage the

availability of formal specifications in various forms [46, 113].

6.2 Fault localization

Fault localization is the process of locating statements that should be changed in order to correct a given fault. Many of the approaches to automated fault localization rely on measures of code coverage or program states.

6.2.1 Code coverage.

Code coverage metrics have been used to rank instructions based on their likelihood of triggering failures. Jones et al. [52], for example, introduce the notion of *failure rate*: an instruction has a high failure rate if it is executed more often in failing test cases than in passing test cases. A block of code is then "suspicious" of being faulty if it includes many instructions with high failure rate; Jones et al. also implemented visualization support for their debugging approach in the tool Tarantula.

Renieris and Reiss's fault localization technique [96] is based on the notion of *nearest neighbor*: given a test suite, the nearest neighbor of a faulty test case t is the passing test case that is most similar to t. Removing all the instructions mentioned in the nearest neighbor from the faulty test produces a smaller set of instructions; instructions in the set are the prime candidates to be responsible for the fault under consideration. Artzi et al. [12] apply similar techniques to rank statements together with their runtime values to locate execution faults in PHP web applications. For better fault localization effectiveness, Artzi et al. also exploit concolic test-generation techniques to build new test cases that are similar to the failing one, the basic idea being that the differences between similar passing and failing test executions highly correlate with the fault cause.

Many other authors have extended code coverage techniques for fault localization. For example, Zhang et al. [112] address the propagation of infected program states; Liu et al. [59] rely on a model-based approach; and Wong et al. [108] perform an extensive comparison of variants of fault localization techniques and outline general principles behind them (which we follow in Section 4.1.2.2). Pytlik et al. [94] discuss the limitations of using only state invariants for fault localization, a limitation that AutoFix avoids by combining snapshots based on state invariants with snapshots based on enumeration (Section 4.1.1).

6.2.2 *Program states.*

The application of code coverage techniques produces a set of instructions likely to be responsible for failure; programmers still have to examine each instruction to understand what the problem is. Fault localization techniques based on program states aim at providing more precise information in such contexts: state-based analyses are finer-grained than those based only on code coverage because they can also report suspicious state values that should be changed. Huang et al [47], for example, suggest to insert check points in the program to mark "points of interest". Then, a dynamic analysis—applied to program states rather than locations—can identify a set of suspicious states; furthermore, the usage of check points introduces the flexibility to skip uninteresting parts of the computation, for example repeated iterations of a loop. *Delta debugging* [110, 111] addresses similar issues: isolating the variables, and their values, relevant to a failure by analyzing the state difference between passing and failing test cases.

Angelina [18] is a technique that repeatedly runs a program against a group of tests with the intent of discovering a list of expressions from the suspected faulty code such that: changing the value of any such expression at runtime could make the failing tests pass, while still letting the originally passing tests pass. Such expressions are then reported to the programmer as suggestions: building the actual corrections is still the programmer's job.

Most fault localization techniques target each fault individually, and hence they perform poorly when multiple bugs interact and must be considered together. To address such scenarios, Liblit et al. [57] introduce a technique that separates the effects of multiple faults and identifies predictors associated with each fault.

While the research on automated fault localization has made substantial progresses, effectively applying fault localization in practice to help programmers still poses open challenges. Parnin and Orso [84] demonstrate that most automated debugging techniques focus on tasks (mostly, localization) that represent only a small part of the real debugging activity. Automated fixing techniques can help in this regard by providing an additional layer of automation that includes synthesizing suitable validated corrections.

6.2.3 *Fault localization for automatic fixing.*

The ImpleFix technique in Chapter 4 includes fault localization techniques (Section 4.1.2). To generate fixes completely automatically fault localization must be sufficiently precise to suggest only a limited number of "suspicious" instructions. In our case, using contracts helps to restrict the search to the boundaries of the routine where a contract-violation fault occurs. Then, we combine dynamic analysis techniques based on those employed for fault localization (Section 4.1.2.2) with

simple static analyses (Section 4.1.2.1) to produce a ranking of state snapshots within routines that is sufficiently accurate for the fixing algorithm to produce good-quality results.

Coker and Hafiz [23] show how to identify, through static analyses based on types, unsafe integer usages in C programs; simple program transformations can automatically patch such unsafe usages.

6.3 Source-code Repairs

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

6.3.1 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. [49] 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 [11, 8] 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. [107, 106] describe GenProg, a technique that uses genetic programming[1] to mutate a faulty program into one that passes all given test cases. GenProg has been extensively evaluated [55, 54] 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. [53] 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 4.1.3); 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

[1] See also Arcuri and Briand's remarks [9, Sec. 2] on the role of evolutionary search in Weimer et al.'s experiments [107].

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 [44, 42, 8], to few benchmarks used for fault localization [49], or do not aim at complete automation [105].

GenProg can fix 52% of 105 bugs with the latest improvements [54]; Par fixes 23% of 119 bugs (GenProg fixes 13% of the same 119 bugs [53]). In our experiments in Section 4.2, we target almost twice as many bugs (204) and AutoFix fixes 42% 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 8 C programs totaling over 5 million lines of code; Par worked on 6 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,[2] 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 2.1.1).

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 [55]. It is therefore hard to compare with AutoFix's. Finally, AutoFix works with re-

[2]GenProg's sensitivity to the design choices of its genetic algorithm has been recently investigated [56].

markably limited computational resources: using the same pricing scheme used in GenProg's evaluation [54][3], AutoFix would require a mere $0.01 per valid fix (computed as $0.184 \times$ total fixing time in hours / total number of valid fixes) and $0.03 per proper fix; or $0.06 per valid and $0.23 per proper fix including the time to generate tests—two orders of magnitude less than GenProg's $7.32 per valid fix.

6.3.2 Axiomatic Reasoning

He and Gupta [44] 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.

6.3.3 Constraint-based Techniques.

Gopinath et al. [42] 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

[3] 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.

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. [75] build on previous work [18] 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 4.5 and 4.7) 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.

6.3.4 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 4.1.3.3).

In previous work, we developed Pachika [28], 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 [105] 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" [105].

6.4 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 [67, 51, 51].

Gorla et al. [17, 43] 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 [16], 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 [48] target atomicity violation errors in concurrent programs, which they fix by introducing synchronization statements automatically. More recently, Jin et al. [50] developed the tool AFix that targets the same type of concurrency errors.

Abraham and Erwig [1] 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 [2].

Samimi et al. [98] 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 4.1.3.4).

6.5 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. [31, 33] provide generic support for dynamic patching inside the Java language.

6.5.1 *Data-structure repair.*

Demsky and Rinard [32] 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 [34] 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 [63, 64], 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 [97] 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 [32], as well as on other operations that are naturally expressed as constraint satisfaction problems.

6.5.2 *Memory-error Repair.*

The ClearView framework [89] dynamically corrects buffer overflows and illegal control flow transfers in binaries. It exploits a variant of Daikon [37] 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 [14, 77] 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.

6.6 Invariant Inference

Invariant inference techniques learn assertions that hold for a given implementation. These techniques are naturally classified in *static* and *dynamic*. Static techniques analyze the source code to infer specification elements. Since inferring all but the simplest classes of properties is undecidable, static techniques are usually sound but incomplete. Abstract interpretation is a fundamental framework for static invariant inference [24], which has been applied in many different contexts.

SpeciFix relies instead on *dynamic* techniques for invariant inference. These summarize properties that are invariant over multiple runs of a program; their advantage over static techniques is that dynamic approaches do not require a sophisticated analytical framework and are applicable to the whole programming language: they work on anything that can be executed. While dynamic techniques provide no guarantees of soundness or completeness, they work quite well in practice. Dynamic invariant inference has been pioneered by the Daikon tool [36]. Daikon uses a pre-defined set of templates describing common relations among program variables. Much work has been done to extend and improve the Daikon approach; for example to support object-oriented features [26], and to infer complex and often complete postconditions [101]. The dynamic approach has also been applied to other kinds of specifications such as finite-state behavioral specifications [5, 27, 61, 109] and algebraic specifications [40, 45].

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

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.

This chapter summarizes our contribution to automatic program fixing as provided in this thesis, and outlines potential directions for future research. Section 7.1 recapitulates the major contributions, and Section 7.2 describes future work.

7.1 Main Contributions

AutoFix provides an automatic and integrated solution to program fixing, which programmers could easily use in development. The solution relies on the presence of simple specification elements in the form of contracts to provide high-quality fix suggestions and to enable automation of the whole debugging process. The most important components in the work of this thesis include the following:

1. The ImpleFix technique, which generates *implementation* fixes to program faults automatically. In an extensive experiment with over 200 faults in software of various quality, ImpleFix generated proper fixes, i.e. fixes that remove the fault under consideration without introducing other faulty or unexpected behavior, to 25% of the faults, requiring 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.

2. The SpeciFix technique, which automatically fixes programming bugs by rectifying *specifications* in the form of simple contracts, and therefore com-

plements the ImpleFix technique. In an experimental evaluation, we ran SpeciFix on 44 bugs of Eiffel standard data-structure libraries, and SpeciFix produced proper fixes to 11 (or 25%) of the bugs, using similar amount of time as ImpleFix requires.

3. The AutoFix tool, which combines ImpleFix and SpeciFix and is integrated into the EVE IDE. The tool presents itself as a recommendation system that automatically finds bugs and suggests fixes to the bugs.

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.

7.2 Future Work

We now propose some future work following the contributions highlighted in the previous section.

Although the previous experiments suggest that simple contracts and automatically generated tests are already useful in automatic fixing, more extensive experiments are necessary to find out how the amount and the quality of contracts in a program as well as the use of manually written tests would influence AutoFix's behavior and results. Also, the experiment in Section 5.2 does not provide enough data points to answer questions related to the performance of SpeciFix with statistical significance. To answer these questions we need to conduct more experiments, e.g., of similar scale as the one described in Section 4.2.

An important goal in the future is to provide a more meaningful ranking of all candidate fixes. With the help of AutoFix, fixing a fault often boils down to identifying a proper candidate from all the generated implementation and specification fixes. The two types of fix candidates are, however, presented to the user in separate groups now, and little information is provided to the user regarding which candidate is more likely to be proper. The new ranking algorithm should take into account the existing ranking mechanism and the nature of different fixes, and it should also learn from the manual inspection results of the candidate fixes as well as the manually written fixes from programmers, so that the ranking will get improved over time.

Automatic fixing would benefit from customized test generation. In its current implementation, AutoFix uses AutoTest as it is to detect faults and prepare test cases. AutoTest was, however, designed to find as many faults as possible in a given time, and generating appropriate passing and failing tests that would lead to optimal automatic fixing results is not one of its concerns. We plan to investigate further how test case selection affects the performance of AutoFix, design heuristics for optimal test case selection for fixing, and use the heuristics to guide test preparation in AutoTest.

The performance of AutoFix can be improved by incorporating more static techniques. AutoFix can already automatically generate candidate fixes using mostly dynamic techniques. By combining the dynamic techniques with other static techniques like constraint solving and axiomatic reasoning we can improve both the effectiveness and the efficiency of AutoFix. For example, constraint solving may provide fixes that are hard or even impossible to get from instantiating the fixing schemas in Section 4.1.4.1, and axiomatic reasoning could help decide whether to skip generating implementation fixes depending on whether the precondition and the postcondition of the routine under debugging conflict with each other and therefore leave no space for satisfactory implementation fixes.

BIBLIOGRAPHY

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

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

[3] Shaukat Ali, Lionel C. Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36(6):742–762, 2010.

[4] Frances E. Allen. Control flow analysis. In *Proceedings of a symposium on Compiler optimization*, pages 1–19. ACM, 1970.

[5] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *POPL*, pages 4–16, 2002.

[6] James H. Andrews, Susmita Haldar, Yong Lei, and Felix Chun Hang Li. Tool support for randomized unit testing. In *Proceedings of the 1st International Workshop on Random Testing*, pages 36–45. ACM, 2006.

[7] James H. Andrews, Tim Menzies, and Felix Chun Hang Li. Genetic algorithms for randomized unit testing. *IEEE Transactions on Software Engineering*, 37(1):80–94, 2011.

[8] Andrea Arcuri. Evolutionary repair of faulty software. *Applied Soft Computing*, 11(4):3494–3514, 2011.

[9] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1–10. ACM, 2011.

[10] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. Formal analysis of the effectiveness and predictability of random testing. In *Proceedings of the 2010 International Symposium on Software Testing and Analysis*, pages 219–230. ACM, 2010.

[11] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 162–168. IEEE, 2008.

[12] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Fault localization for dynamic web applications. *IEEE Transactions on Software Engineering*, 38(2):314–335, 2012.

[13] Kent Beck. *Test-Driven Development*. Addison-Wesley, 2002.

[14] Emery D. Berger. Software needs seatbelts and airbags. *Communications of the ACM*, 55(9):48–53, 2012.

[15] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[16] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolò Perino, and Mauro Pezzè. Automatic recovery from runtime failures. In *Proceedings of the 35th International Conference on Software Engineering*, pages 782–791. IEEE Press, 2013.

[17] Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè. Automatic workarounds for web applications. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 237–246, 2010.

[18] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 121–130. ACM, 2011.

[19] Ya Lun Chou. *Statistical analysis*. Holt, Rinehart and Winston, 1975.

[20] Ilinca Ciupa, Alexander Pretschner, Manuel Oriol, Andreas Leitner, and Bertrand Meyer. On the number and nature of faults found by random testing. *Softw. Test., Verif. Reliab.*, 21(1):3–28, 2011.

[21] Ilinca Ciupa, Alexander Pretschner, Manuel Oriol, Andreas Leitner, and Bertrand Meyer. On the number and nature of faults found by random testing. *Software Testing, Verification and Reliability*, 21(1):3–28, 2011.

[22] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *ACM SIGPLAN Notices*, pages 268–279. ACM Press, 2000.

[23] Zack Coker and Munawar Hafiz. Program transformations to fix C integers. In *Proceedings of the 35th International Conference on Software Engineering*, pages 792–801. IEEE Press, 2013.

[24] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

[25] Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[26] Christoph Csallner and Yannis Smaragdakis. Dynamically discovering likely interface invariants. In *ICSE*, pages 861–864, 2006.

[27] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with ADABU. In *WODA*, pages 17–24, 2006.

[28] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. Generating fixes from object behavior anomalies. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 550–554. IEEE, 2009.

[29] Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 433–436. ACM, 2007.

[30] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer-Verlag, 2008.

[31] Brian Demsky and Alokika Dash. Bristlecone: A language for robust software systems. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, volume 5142 of *Lecture Notes in Computer Science*, pages 490–515. Springer, 2008.

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

[33] Brian Demsky and Sivaji Sundaramurthy. Bristlecone: Language support for robust software applications. *IEEE Transaction on Software Engineering*, 37(1):4–23, 2011.

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

[35] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, pages 213–224. ACM, 1999.

[36] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):99–123, 2001.

[37] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transaction on Software Engineering*, 27(2):99–123, 2001.

[38] H.-Christian Estler, Carlo A. Furia, Martin Nordio, Marco Piccioni, and Bertrand Meyer. Contracts in practice. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *Proceedings of the 19th International Symposium on Formal Methods (FM)*, volume 8442 of *Lecture Notes in Computer Science*, pages 230–246. Springer, May 2014.

[39] EVE: the Eiffel Verification Environment. `http://se.inf.ethz.ch/research/eve/`.

[40] Carlo Ghezzi, Andrea Mocci, and Mattia Monga. Synthesizing intensional behavior models by graph transformation. In *ICSE*, pages 430–440, 2009.

[41] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[42] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. Specification-based program repair using SAT. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 173–188. Springer-Verlag, 2011.

[43] Alessandra Gorla, Mauro Pezzè, Jochen Wuttke, Leonardo Mariani, and Fabrizio Pastore. Achieving cost-effective software reliability through self-healing. *Computing and Informatics*, 29(1):93–115, 2010.

[44] Haifeng He and Neelam Gupta. Automated debugging using path-based weakest preconditions. In *Fundamental Approaches to Software Engineering*, volume 2984 of *Lecture Notes in Computer Science*, pages 267–280. Springer, 2004.

[45] Johannes Henkel, Christoph Reichenbach, and Amer Diwan. Discovering documentation for Java container classes. *IEEE TSE*, 33(8):526–543, 2007.

[46] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy A. Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2), 2009.

[47] Tai-Yi Huang, Pin-Chuan Chou, Cheng-Han Tsai, and Hsin-An Chen. Automated fault localization with statistically suspicious program states. In *Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 11–20. ACM, 2007.

[48] Muhammad U. Janjua and Alan Mycroft. Automatic corrections to safety violations in programs. In *Proceedings of the Thread Verification Workshop*, pages 111–116, 2006.

[49] Dennis Jeffrey, Min Feng, Neelam Gupta, and Rajiv Gupta. BugFix: a learning-based tool to assist developers in fixing bugs. In *Proceedings of the 17th IEEE International Conference on Program Comprehension*, pages 70–79, 2009.

[50] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 389–400. ACM, 2011.

[51] Barbara Jobstmann, Stefan Staber, Andreas Griesmayer, and Roderick Bloem. Finding and fixing faults. *Journal of Computer and System Sciences*, 78(2):441–460, 2012.

[52] James A. Jones, Mary J. Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, pages 467–477. ACM, 2002.

[53] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering*, pages 802–811. IEEE Press, 2013.

[54] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, pages 3–13. IEEE, 2012.

[55] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Transaction on Software Engineering*, 38(1):54–72, 2012.

[56] Claire Le Goues, Westley Weimer, and Stephanie Forrest. Representations and operators for improving evolutionary software repair. In *Genetic and Evolutionary Computation Conference, GECCO '12, Philadelphia, PA, USA, July 7–11, 2012*, pages 959–966. ACM, 2012.

[57] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–26. ACM, 2005.

[58] Benjamin Robert Liblit. *Cooperative Bug Isolation*. PhD thesis, University of California, Berkeley, December 2004.

[59] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. SOBER: statistical model-based bug localization. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 286–295. ACM, 2005.

[60] Lisa Ling Liu, Bertrand Meyer, and Bernd Schoeller. Using contracts and Boolean queries to improve the quality of automatic test generation. In *Proceedings of the 1st International Conference on Tests and Proofs*, volume 4454 of *Lecture Notes in Computer Science*, pages 114–130. Springer-Verlag, 2007.

[61] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *ICSE*, pages 501–510, 2008.

[62] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering*, pages 416–426, 2007.

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

[64] Muhammad Zubair Malik, Junaid Haroon Siddiqui, and Sarfraz Khurshid. Constraint-based program debugging using data structure repair. In *Proceedings of the IEEE Fourth International Conference on Software Testing, Verification and Validation*, pages 190–199. IEEE, 2011.

[65] Matias Martinez and Martin Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 2013.

[66] E. Michael Maximilien and Laurie Williams. Assessing test-driven development at IBM. In *ICSE*, pages 564–569, 2003.

[67] Wolfgang Mayer and Markus Stumptner. Evaluating models for model-based debugging. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 128–137, 2008.

[68] Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification & Reliability*, 14(2):105–156, 2004.

[69] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 2000.

[70] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. Programs that test themselves. *IEEE Computer*, 42(9):46–55, 2009.

[71] Martin Monperrus. A critical review of "automatic patch generation learned from human-written patches": essay on the problem statement and the evaluation of automatic software repair. In *ICSE*, pages 234–242. ACM, 2014.

[72] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[73] Matthias Müller, Rainer Typke, and Oliver Hagner. Two controlled experiments concerning the usefulness of assertions as a means for programming. In *International Conference on Software Maintenance (ICSM)*, October 2002.

[74] Nachiappan Nagappan, E. Michael Maximilien, Thirumalesh Bhat, and Laurie Williams. Realizing quality improvement through test driven development: results and experiences of four industrial teams. *ESE*, 13:289–302, 2008.

[75] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering*, pages 772–781. IEEE Press, 2013.

[76] Martin Nordio, Carlo Ghezzi, Bertrand Meyer, Elisabetta Di Nitto, Giordano Tamburrelli, Julian Tschannen, Nazareno Aguirre, and Vidya Kulkarni. Teaching software engineering using globally distributed projects: the DOSE course. In *Proceedings of Collaborative Teaching of Globally Distributed Software Development – Community Building Workshop*. ACM, 2011.

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

[78] Catherine Oriat. Jartege: a tool for random generation of unit tests for Java classes. Technical Report RR-1069-I, CNRS, 2004.

[79] Manuel Oriol and Sotirios Tassis. Testing .NET code with YETI. In *Proceedings of the 15th IEEE International Conference on Engineering of Complex Computer Systems*, pages 264–265, 2010.

[80] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proceedings of the 19nd European Conference on Object-Oriented Programming*, pages 504–527, 2005.

[81] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, pages 75–84. IEEE, 2007.

[82] Kai Pan, Sunghun Kim, and E. James Whitehead, Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, June 2009.

[83] Parasoft Corporation. Jtest. `http://www.parasoft.com/`.

[84] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 20th International Symposium on Software Testing and Analysis*, pages 199–209. ACM, 2011.

[85] Yu Pei, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Automatic program repair by fixing contracts. In Stefania Gnesi and Arend Rensink, editors, *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 8411 of *Lecture Notes in Computer Science*, pages 246–260. Springer, April 2014.

[86] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. *IEEE Transactions on Software Engineering*, 40(5):427–449, 2014.

[87] Yu Pei, Yi Wei, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Code-based automated program fixing. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 392–395. ACM, 2011.

[88] John Penix. Large-scale test automation in the cloud. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *Proceedings of the 34th International Conference on Software Engineering*, page 1122. IEEE, 2012.

[89] Jeff H. Perkins, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, Martin Rinard, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, and Stelios Sidiroglou. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 87–102, 2009.

[90] Nadia Polikarpova. EiffelBase2. `http://dev.eiffel.com/EiffelBase2`, 2012.

[91] Nadia Polikarpova, Ilinca Ciupa, and Bertrand Meyer. A comparative study of programmer-written and automatically inferred contracts. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 93–104. ACM, 2009.

[92] Nadia Polikarpova, Carlo A. Furia, and Bertrand Meyer. Specifying reusable components. In Gary T. Leavens, Peter O'Hearn, and Sriram Rajamani, editors, *Proceedings of the 3rd International Conference on Verified Software: Theories, Tools, and Experiments*, volume 6217 of *Lecture Notes in Computer Science*, pages 127–141. Springer, August 2010.

[93] Nadia Polikarpova, Carlo A. Furia, Yu Pei, Yi Wei, and Bertrand Meyer. What good are strong specifications? In David Notkin, Betty H.C. Cheng, and Klaus Pohl, editors, *Proceedings of the 35rd International Conference on Software Engineering*, pages 257–266. ACM, May 2013.

[94] Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven P. Reiss. Automated fault localization using potential invariants. In *Proceedings of the 5th International Workshop on Automated and Algorithmic Debugging*, pages 273–276, 2003.

[95] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 191–201. ACM, 2013.

[96] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE/ACM International Conference on Automated Software Engineering*, pages 30–39. IEEE, 2003.

[97] Hesam Samimi, Ei Darli Aung, and Todd D. Millstein. Falling back on executable specifications. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP)*, volume 6183 of *Lecture Notes in Computer Science*, pages 552–576. Springer, 2010.

[98] Hesam Samimi, Max Schäfer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. Automated repair of HTML generation errors in PHP applications using string constraint solving. In *Proceedings of the International Conference on Software Engineering*, pages 277–287. IEEE Press, 2012.

[99] Rohan Sharma, Milos Gligoric, Andrea Arcuri, Gordon Fraser, and Darko Marinov. Testing container classes: random or systematic? In *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering: Part of the joint European Conferences on Theory and Practice of Software*, volume 6603 of *Lecture Notes in Computer Science*, pages 262–277. Springer-Verlag, 2011.

[100] Paolo Tonella. Evolutionary testing of classes. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*, pages 119–128. ACM, 2004.

[101] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. Inferring better contracts. In *ICSE*, pages 191–200. ACM, 2011.

[102] Yi Wei, Serge Gebhardt, Manuel Oriol, and Bertrand Meyer. Satisfying test preconditions through guided object selection. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, pages 303–312, 2010.

[103] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, pages 61–72. ACM, 2010.

[104] Yi Wei, Hannes Roth, Carlo A. Furia, Yu Pei, Alexander Horton, Michael Steindorfer, Martin Nordio, and Bertrand Meyer. Stateful testing: Finding more errors in code and contracts. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 440–443. ACM, 2011.

[105] Westley Weimer. Patches as better bug reports. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, pages 181–190. ACM, 2006.

[106] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109–116, 2010.

[107] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the IEEE 31st International Conference on Software Engineering*, pages 364–374, 2009.

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

[109] Tao Xie, Evan Martin, and Hai Yuan. Automatic extraction of abstract-object-state machines from unit-test executions. In *ICSE*, pages 835–838. IEEE, 2006.

[110] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 1–10. ACM, 2002.

[111] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transaction on Software Engineering*, 28(2):183–200, 2002.

[112] Zhenyu Zhang, W. K. Chan, T. H. Tse, Bo Jiang, and Xinming Wang. Capturing propagation of infected program states. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 43–52. ACM, 2009.

[113] Daniel M. Zimmerman and Rinkesh Nagmoti. JMLUnit: The next generation. In *Proceedings of the 1st International Conference on Formal Verification of Object-Oriented Software*, volume 6528 of *Lecture Notes in Computer Science*, pages 183–197. Springer, 2010.