Automated Program Repair in an Integrated Development Environment

Yu Pei, Carlo A. Furia, Martin Nordio, Bertrand Meyer
Chair of Software Engineering, Department of Computer Science, ETH Zurich, Switzerland
Email: firstname.lastname@inf.ethz.ch

***Abstract*—We present the integration of the AutoFix automated program repair technique into the EiffelStudio Development Environment. AutoFix presents itself like a recommendation system capable of automatically finding bugs and suggesting fixes in the form of source-code patches. Its performance suggests usage scenarios where it runs in the background or during work interruptions, displaying fix suggestions as they become available. This is a contribution towards the vision of semantic Integrated Development Environments, which offer powerful automated functionality within interfaces familiar to developers. A screencast highlighting the main features of AutoFix can be found at: http://youtu.be/Ff2ULiyL-80.**

I. Introduction

That programs have bugs[1] may well be a platitude, but it's one whose consequences cost billions every year in manual repairs and maintenance. Any tool that can support even partial automation has the potential to significantly help reduce the surging costs of repairing defective software.

Tools based on *syntactic* techniques—mainly type checking—which can suggest simple corrections to straightforward compile-time errors have become standard components of modern Integrated Development Environments (IDEs) in the form of widely used functionalities such as Eclipse's Quick Fix. Only in the last few years, however, have the first fully automatic *semantic* techniques become available [1]–[3] that can fix bugs occurring at *runtime* such as overflows, null pointer dereference, and even functional errors—providing genuine *automatic program repair*. The bulk of research in automated program repair (Sect. IV) has focused on demonstrating feasibility of the proposed techniques on realistic programs and on evaluating performance and limitations. Prototype tools implementing these techniques largely remain stand-alone applications usable only by expert researchers. However, if automated program repair has to make a wider impact, it should become well integrated as an unobtrusive IDE component that programmers can routinely use without in-depth knowledge of the techniques behind it.

This paper describes the results of integrating our automated repair technique, called AutoFix and introduced in previous work [4], [5], into the EiffelStudio development environment. The resulting integrated tool, which we also call AutoFix, is available within the IDE in very much the same way as any other functionality such as compilation or refactoring.

Work partially supported by ERC grant CME/291389; by SNF grants 200020-134974 and 200021-134976; and by Hasler-Stiftung grant #2327.

[1]In the paper, we use the terms "error", "fault", and "bug" as synonyms.

To maximize the degree of automation and simplify user interaction, AutoFix leverages the simple *contracts* (pre- and postconditions and class invariants) that Eiffel programmers normally provide in addition to imperative code. While writing contracts is a well-established practice among Eiffel programmers [6], AutoFix users can still supply manually written tests to complement those generated automatically. This gives more flexibility to experts without making the learning curve steep for users new to AutoFix, who in fact can benefit from it without any knowledge of its techniques.

Like most techniques for automated program repair, AutoFix mainly uses *dynamic analysis* (i.e., running tests) to collect information about program behavior and to validate repairs. Dynamic analysis tends to be significantly slower than purely static analysis, and hence AutoFix cannot produce fix suggestions in real time as the user types. However, AutoFix's performance is still compatible with usage scenarios where automated repair runs in the background on the classes of a project that are not currently being edited, or on the whole project while the developer is on a break or off for the day. As they come back, the IDE will show a selection of fix suggestions heuristically ranked by relevance and simplicity.

Previous evaluations of AutoFix (Sect. III-C), whose results are now reproducible within EiffelStudio, indicate that programmers can use a serviceable program repair technique as part of their normal development process without disruptive changes or specific training. This is a step towards achieving the vision of semantic IDEs [3], which empower programmers with a variety of powerful tools under a unified interface.

AutoFix's project page http://se.inf.ethz.ch/research/autofix/ includes an overview of the technique, a user manual/tutorial, source and pre-compiled binaries, a virtual machine image, and a demo screencast.

The bulk of the paper focuses on describing AutoFix in EiffelStudio from a user's perspective (Sect. II). Sect. III provides a brief summary of AutoFix's fixing techniques and of extensive experimental evaluations in previous work.

II. Using AutoFix

A. A Usage Scenario

We give a succinct overview of AutoFix in EiffelStudio from the perspective of a nondescript user named Max.

As Max 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 1a) looks familiarly similar to other tools already available in EiffelStudio that Max 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, Max 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, Max launches AutoFix on `MY_LIST` with default settings and leaves for lunch.

When she's back after one hour, Max sees that automatic testing (ran by AutoFix using contracts as oracles) has found a bug in `MY_LIST`'s copy method `duplicate` (Figure 1b). In a different tab (Figure 1c), 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, Max 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.[2] All the four fixes have been validated against the available tests, but Max 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, Max opts for deploying the implementation fix that handles the empty list case separately. This requires one click.

AutoFix has helped Max 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. Max plans to start another, longer AutoFix session when she'll leave in the evening.

B. AutoFix in Practice: Features and Usage

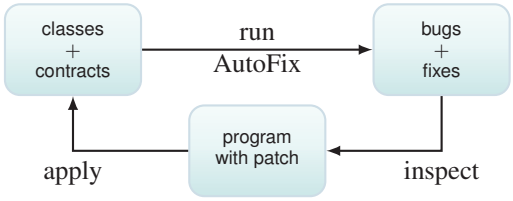


Fig. 2: Usage workflow of AutoFix in EiffelStudio.

AutoFix finds bugs and suggests corrections in the form of source code patches that, if applied, permanently remove incorrect behavior. AutoFix works on Eiffel programs annotated with simple contracts such as pre- and postconditions.

Running AutoFix. In the typical usage workflow, users can launch AutoFix at any time on one or more classes of the current project (top left box in Figure 2). To this end, the *settings* tab (Figure 1a) inputs the name of one or more classes to be analyzed—the only required input to AutoFix. Optionally, users can change the default values of a few other *settings* such as for how long testing and fixing should proceed, how many valid fixes should be displayed, and whether AutoFix should try to fix implementation (i.e., imperative code), specification (i.e., contracts), or both.

Bugs. AutoFix takes advantage of the contracts that Eiffel programmers normally write as part of the project they develop. The first usage of contracts is as automated *testing oracles*. AutoFix calls AutoTest, another component of EiffelStudio, in the background to generate unit tests for the classes under analysis. AutoTest executes methods on random class instances and classifies each execution as passing or failing: since contracts are Boolean expressions that can be evaluated at runtime, a test such that all assertions on its execution path evaluate to true is passing; otherwise (some assertion evaluates to false) the test is failing. For example, a routine terminating in a state where the postcondition evaluates to false exposes a bug—an inconsistency between specification and implementation. As AutoTest discovers new faults, AutoFix displays them in real time in the *faults* tab (Figure 1b).

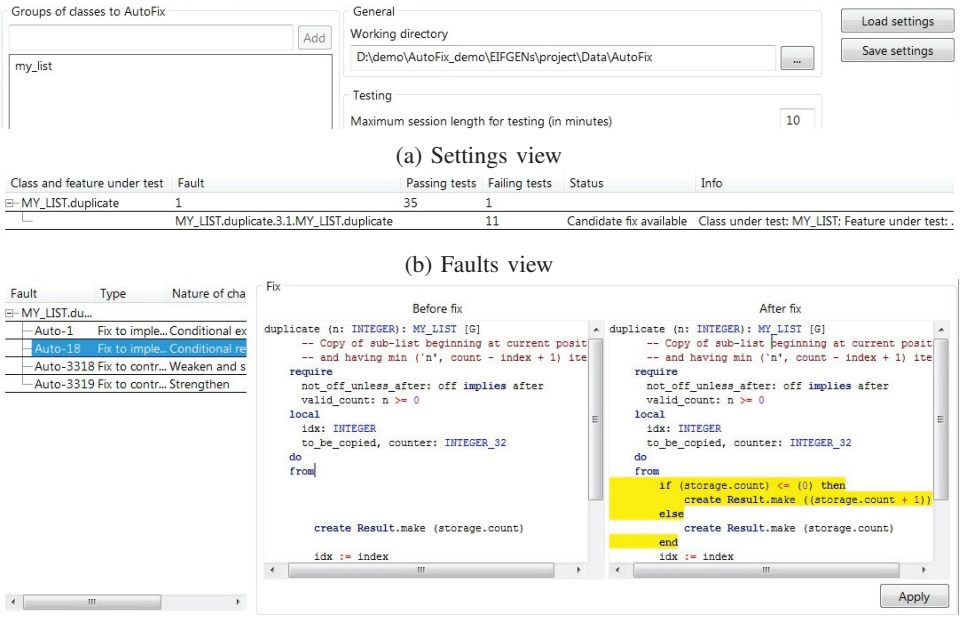
Fixes. As soon as the time allotted to testing expires (10 minutes by default), AutoFix starts analyzing the generated tests. A collection of passing and failing tests for the same method such that all failing tests violate the same contract characterizes incorrect (failing tests) as well as correct (passing tests) behavior for the same program element. Using dynamic analysis, AutoFix gathers information about which locations are more likely to be responsible for the bug, and explores modifications to the program that would turn all failing tests into passing ones, while still passing all previously passing tests. Two fixing algorithms are at work in AutoFix. One algorithm [4] suggests changes to the *implementation* without changing the available contracts; another algorithm [5] suggests changes to the *specification* without changing the implementation (see Sect. III). The fix suggestions generated by the two algorithms are validated against all available tests for the bug being analyzed. Only those that pass all of them (including, crucially, those that previously failed) are *ranked* and reported to users in the *fixes* tab (Figure 1c; and top right box in Figure 2).

Ranking in AutoFix uses heuristics that estimate the usefulness and impact of a fix (see Sect. III). All else being equal, fixes that are closer to the locations suspected of originating a fault and that are more general are ranked higher. For example, a fix that adds an instruction just before the location of a fault tends to be preferred over another fix for the same fault that adds an instruction at the entry point of a method; and a fix that relaxes a precondition tends to be preferred over another fix that strengthens it (thus possibly breaking unforeseen clients).[3]

Inspecting and applying fixes. Ultimately, users are responsible for choosing and deploying fix suggestions. To this end, AutoFix offers a diff view in its *fixes* tab (Figure 1c).

[2]The bug in this example reproduces a real fault found in the Base data-structure library distributed with EiffelStudio.

[3]We plan to extend the current UI to include information used in the ranking to help users select among multiple fixes.



(a) Settings view

(b) Faults view

(c) Fixes view

Fig. 1: From top to bottom: AutoFix's *settings*, *faults*, and *fixes* views in EiffelStudio.

Users select a fix from the list, and its effect is displayed (bottom box in Figure 2) by showing the relevant portions of the codebase before and after applying the fix. If a fix is found satisfactory, users can *apply* it to the actual codebase, which changes implementation or specification as suggested.[4]

III. How AutoFix Works

We described the details of how AutoFix works in previous work [2], [4], [5]. Here, we briefly summarize its algorithms, architecture, and experimental evaluation.

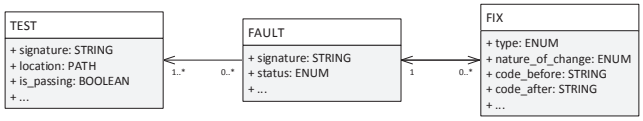
A. Fixing Algorithms

Figure 3 gives an idea of the two main fixing algorithms integrated in AutoFix. One algorithm (top row) suggests fixes to implementations. It uses dynamic analysis to identify "snapshots" (abstract program states) that are likely to be responsible for a fault; it synthesizes some fixing snippets that change the suspicious snapshots' state; and it builds fix suggestions (candidate fixes) by injecting the snippets into the program at failing locations. Validation uses all available tests, and ranks fixes that pass validation according to their suspiciousness score determined by dynamic analysis. Another algorithm (bottom row) suggests fixes to specifications. 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. Validation uses all available tests, and ranks fixes that pass validation by preferring weaker (less restrictive) to stronger ones.

[4]Applying specification fixes requires choosing where in the inheritance hierarchy they should be applied—a choice currently left to users.

B. Architecture and Implementation

AutoFix integrates into EiffelStudio using the model-view-controller pattern. The *model* consists of classes `TEST`, `FAULT`, and `FIX`: each fault is associated with a list of passing and failing tests; each fixing session targets a particular fault and proposes a collection of candidate fixes as outcome. Output features in two *views*: the *faults* view (Figure 1b) and the *fixes* view (Figure 1c). AutoFix's *controller* is responsible for coordinating testing and fixing engines, and for collecting results and updating model and views.



AutoFix is integrated in the research branch of the EiffelStudio IDE and works on programs written in Eiffel. Its concepts and techniques are applicable to any IDE for programming languages supporting annotations (e.g., Java's JML or .NET's CodeContracts).

C. Experimental Evaluation

The following table summarizes the experimental evaluation we conducted in previous work, which applied AutoFix to a significant number of bugs from various Eiffel codebases.

fix	LOC	faults	valid	proper	testing	fixing
implem.	72,920	204	42%	25%	17 min	3 min
spec.	24,608	44	95%	25%	31 min	3 min

AutoFix processed over 200 unique faults discovered by automatic random testing. It suggested fixes to the implementation [4] for 42% of these faults (top data row); 25% of

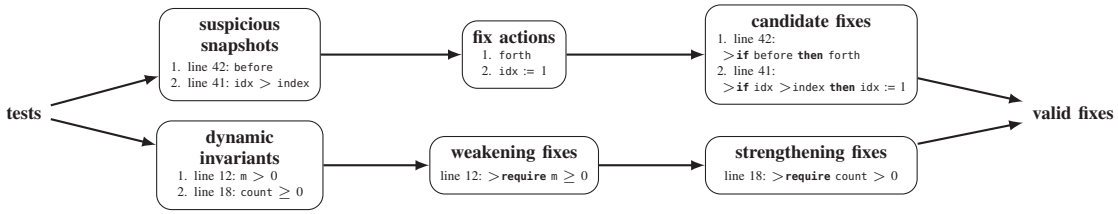


Fig. 3: An overview of the AutoFix algorithms that generate implementation fixes (top row) and contract fixes (bottom row).

the fix suggestions are *proper*, that is completely satisfactory according to manual inspection; on average, AutoFix takes just 3 minutes per valid fix (after an average automatic testing session of 17 minutes). A smaller evaluation of AutoFix's specification fixes [5] targeted 44 unique faults (bottom data row); it suggested fixes to the contracts for 95% of them; while the majority of them are too restrictive to be deployed as is, 25% of the fix suggestions are *proper*, and often preferable to implementation fixes of the same faults; on average, AutoFix takes just 3 minutes per valid fix (after an average automatic testing session of 31 minutes).

IV. Related Work & Limitations of AutoFix

While syntactic error correction techniques have long been widely available, only in the last few years have the first effective program repair techniques been developed. Following GenProg's pioneering results [1], automated program repair has quickly become a burgeoning area of research, producing a variety of approaches and tools, e.g., [3], [7]–[10].

With the exception of SIDE [3], no other automated program repair techniques are integrated in an IDE. Part of the reason may be performance: state-of-the-art repair techniques are based on computationally expensive heuristic runtime search (e.g., genetic algorithm), which leads to running times incompatible with interactive usage. AutoFix's performance is lagging compared to SIDE's purely static analysis, but it's such that it is feasible to let it run in the background or during breaks: as mentioned in Sect. III-C, generating fixes often requires only few minutes. On the other hand, using dynamic techniques such as AutoFix's has advantages over static analysis in terms of no false positives, no loss of precision, and great flexibility in the kinds of programs that can be analyzed—anything that can be executed.

Using contracts helps abstract program executions, and hence ultimately helps AutoFix's performance; but also limits the applicability of AutoFix's techniques. In the context of integration into an IDE for Eiffel, however, requiring contracts is not a real limitation, as writing them is an accepted practice among Eiffel developers [6] and AutoFix works with the simple, incomplete contracts that are normally available.

Fix *quality* is an aspect largely neglected [11] in the research on program repair. AutoFix's work is an exception, as we introduced the notion of *proper* fix [12] to characterize those that can be deployed fully satisfactorily. AutoFix's ranking heuristics help programmers winnow out improper fixes, but they are not infallible. Better heuristics would be particularly useful for specification fixes, which tend to include more validated but improper fixes (Sect. III-C). Developing better classifiers of high-quality fixes belongs to future work, where we plan to abstract from human assessment criteria [8].

Other limitations of AutoFix, following from limitations of components in its toolchain and from the supported fixing actions, are discussed in [4], [5].

V. Conclusions

We described the results of integrating the AutoFix automated repair technique into the EiffelStudio development environment—a contribution to the vision of *semantic* IDE [3]. Future work will improve the user experience by providing better classification of high-quality fixes; by integrating different automatic testing techniques with better performance; and by combining fixing with static checking.

References

[1] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *ICSE*. IEEE, 2009, pp. 364–374.

[2] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *ISSTA*. ACM, 2010, pp. 61–72.

[3] F. Logozzo and T. Ball, "Modular and verified automatic program repair," in *OOPSLA*. ACM, 2012, pp. 133–146.

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

[5] Y. Pei, C. A. Furia, M. Nordio, and B. Meyer, "Automatic program repair by fixing contracts," in *FASE*, ser. LNCS, vol. 8411. Springer, 2014, pp. 246–260.

[6] H.-C. Estler, C. A. Furia, M. Nordio, M. Piccioni, and B. Meyer, "Contracts in practice," in *FM*, ser. LNCS, vol. 8442. Springer, 2014, pp. 230–246.

[7] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *ICSE*. IEEE, 2012, pp. 3–13.

[8] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *ICSE*. IEEE, 2013, pp. 802–811.

[9] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: program repair via semantic analysis," in *ICSE*. IEEE Press, 2013, pp. 772–781.

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

[11] M. 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*. ACM, 2014, pp. 234–242.

[12] Y. Pei, Y. Wei, C. A. Furia, M. Nordio, and B. Meyer, "Code-based automated program fixing," in *ASE*. ACM, 2011, pp. 392–395.