Code-Based Automated Program Fixing

Yu Pei · Yi Wei · Carlo A. Furia · Martin Nordio · Bertrand Meyer
Chair of Software Engineering, ETH Zurich, Switzerland firstname.lastname@inf.ethz.ch

***Abstract*—Initial research in automated program fixing has generally limited itself to specific areas, such as data structure classes with carefully designed interfaces, and relied on simple approaches. To provide high-quality fix suggestions in a broad area of applicability, the present work relies on the presence of contracts in the code, and on the availability of static and dynamic analyses to gather evidence on the values taken by expressions derived from the code. The ideas have been built into the AutoFix-E2 automatic fix generator. Applications of AutoFix-E2 to general-purpose software, such as a library to manipulate documents, show that the approach provides an improvement over previous techniques, in particular purely model-based approaches.**

***Keywords*-automated debugging and fixing; program synthesis**

I. Introduction

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 providing completely automated debugging in some situations. This has been achieved with the combination of several techniques developed independently: automated testing to detect errors, fault localization to locate instructions responsible for the errors, and dynamic analysis to choose suitable corrections among those applicable to the faulty instructions.

A few premises make such automated debugging techniques work in practice. First, the majority of errors in programs admit simple fixes [1]; second, the availability of contracts (pre and postconditions, class invariants) can dramatically improve the accuracy of both error detection and fault localization.

Our previous work in this area [2], [3] takes advantage of these observations to perform an analysis of faults in object-oriented programs with contracts and correct them. The analysis constructs an abstract model of correct and incorrect executions, which summarizes the information about the program state at various locations in terms of state invariants. The invariants express the values returned by *public queries* (functions) of a class—the same functions used by developers in the contracts that document the implementation. The comparison of the invariants characterizing correct and incorrect runs suggests how to fix errors: whenever the state signals the "incorrect invariant", execute actions to avoid triggering the error. A behavioral model of the class, also relying on state invariants, suggests the applicable "recovery" actions. We call this approach to automated program fixing *model-based*, given that a model, based on state invariants, abstracts the correct and incorrect visible behavior.

The efficacy of model-based fixing fundamentally depends on the quality of the public interfaces, because invariants are mostly based on public queries. The present paper introduces a more general approach to automated fixing which works successfully even for classes with few public queries. The approach is still based on the dynamic analysis of correct and incorrect runs. However, rather than merely monitoring the value of queries, the analysis proactively gathers evidence in terms of values taken by *expressions* appearing in the program text. An algorithm built upon fault localization techniques—based on static and dynamic analysis—ranks expressions and their values according to their likelihood of being indicative of error. The expressions ranking highest are prime candidates to guide the generation of fixes: when an expression takes a "suspicious" value, execute actions that change the value to "unsuspicious". We call this novel approach *code-based* to designate the white-box search for information denoting faults in the program text; code-based techniques, however, also exploit information in the form of state invariants and public queries to reproduce the results of model-based techniques when these are successful.

We implemented code-based fixing in the tool AutoFix-E2. The experiments in Section IV demonstrate that code-based techniques can automatically fix more errors than model-based approaches, even beyond data structure implementations—the natural target of model-based and random-testing techniques, for their rich public interfaces.

This paper is a short summary that focuses on the results and illustrates them with a few examples; an extended version with details, and including an analysis of related work, is available as technical report [4].

II. Automated Fixing: an example

The EiffelBase class *TWO_WAY_SORTED_SET* implements a set data structure with a doubly-linked list. An internal cursor *index* (an integer attribute) is useful to navigate the content of the set: the actual elements occupy positions 1 to *count* (another integer attribute, storing the total number of elements in the set), whereas the indexes 0 and *count* + 1 correspond to the positions *before* the first element and *after* the last. Listing 1 shows the routine *move_item* of this class, which takes an argument *v* of generic type *G* that must be a reference to an element already stored in the set; the routine then moves *v* from its current (unique) position in the set to the immediate left of the internal cursor *index*. The routine's precondition (**require**) formalizes the constraint on the input. After saving the cursor position as the local variable *idx*, the loop in lines 7–10 performs a linear search for the element *v* using the internal

Listing 1. Routines of *TWO_WAY_SORTED_SET*.

```
1  move_item (v: G)
2      -- Move 'v' to the left of cursor.
3    require v ≠ Void ; has (v)
4    local idx: INTEGER ; found: BOOLEAN
5    do
6      idx := index
7      from start until found or after loop
8            found := (v = item)
9            if not found then forth end
10     end
11     check found and not after end
12     remove
13     go_i_th (idx)
14     put_left (v)
15   end
16
17 go_i_th (i: INTEGER) require 0 ≤ i ≤ count + 1
18  put_left (v: G) require not before
19 before: BOOLEAN do Result := (index = 0) end
```

cursor: when the loop terminates, *index* denotes *v*'s position in the set. The three routine calls on lines 12–14 complete the work: *remove* takes *v* out of the set; *go_i_th* restores *index* to its original value *idx*; *put_left* puts *v* back in the set to the left of the position *index*.

AutoTest [5] reveals, completely automatically, a subtle error in this implementation of *move_item*, due to the fact that calling *remove* decrements the *count* of elements in the set by one. AutoTest produces a test that calls *move_item* when *index* equals *count* + 1; after *v* is removed, this value is not a valid position because it exceeds the new value of *count* by two, while a valid cursor ranges between 0 and *count* + 1; hence, *go_i_th*'s precondition (line 17), which enforces the consistency constraint on *index*, is violated on line 13.

Code-Based Fixing at Work. The fault revealed by the test is actually a special case of a more general error which occurs whenever *v* appears in the set in a position to the left of the initial value of *index*: even if *index* ≤ *count* initially, *put_left* will insert *v* in the wrong position as a result of *remove* decrementing *count*—which indirectly shifts the index of every element after *index* to the left by one. AutoFix-E2 can completely correct the error, beyond the specific case reported by the failed test: it builds the expression *idx* > *index* to characterize the error state and generates the following patch introduced before line 13:

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

which re-scales *idx* to reflect the fact that the object in position *idx* has been shifted left.

Model-Based Fixing at Work. Model-based techniques can correct the error in the invocation of *go_i_th*, only for the specific instance exposed by the test case where *index* = *count* + 1, that is when *after* holds. Based on this, a possible partial fix consists in adding **if** *after* **then** *back* **end** as first instruction on line 5. This fix is not only partial but also unlikely to be generated in practice, because it modifies code which is several instructions away from where the contract violation occurs, but AutoFix-E2's heuristics favor fixes that are local to restrict the search space. As shown above, code-based techniques do not suffer these limitations.

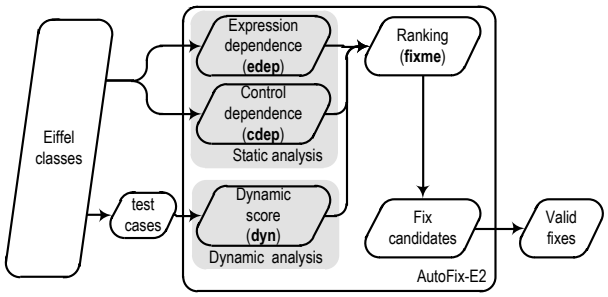

Fig. 1. How code-based fixing works.

III. Code-Based Fixing

Figure 1 depicts the main steps of the code-based fixing process. Code-based fixing works on Eiffel classes equipped with contracts [6]: preconditions, postconditions, and class invariants; each contract element consists of one or more *clauses*. The contracts of a class provide a way to determine functional *errors* in the implementation.

Test-Case Generation. Every session of code-based fixing starts by collecting information about the runtime behavior of the routine under fix. The raw form of such information is a collection of *test cases*; each test case is *passing* if it does not violate any contract and *failing* otherwise. AutoFix-E2 generates test cases with AutoTest [5], developed in our previous work; the use of AutoTest makes the fixing process in AutoFix-E2 completely automatic.

Predicates, Expressions, and States. Evidence of faults takes the form of boolean *predicates*, built by combining *expressions* extracted from the program text and the violated contract clause. The evaluation of a predicate p at a program location ℓ gives a *component* $\langle \ell, p, v \rangle$ of the program state at that location, where v is the value of p at ℓ.

Code-based fixing collects the components from all the test cases and ranks the components so that a triple $\langle \ell, p, v \rangle$ appearing high in the ranking indicates that an error is likely to have its origin at location ℓ when predicate p evaluates to v. Correspondingly, the fixes generated automatically try to change the usage of p at ℓ whenever p's value is v.

Static Analysis. Static analysis extracts evidence from the program text independently of the runtime behavior, in the form of two scores: control dependence and expression dependence.

Control dependence measures the distance, in terms of number of instructions, between program locations. For two program locations ℓ_1, ℓ_2 in a routine r, the *control distance* $\mathsf{cdist}(\ell_1, \ell_2)$ is the length of the shortest directed path from ℓ_1 to ℓ_2 on the control-flow graph. Then, the *control dependence*

$\mathsf{cdep}(\ell, \jmath)$ is the normalized score:

$$\mathsf{cdep}(\ell, \jmath) \;=\; 1 - \frac{\mathsf{cdist}(\ell, \jmath)}{\max_\lambda \mathsf{cdist}(\lambda, \jmath)}$$

where λ ranges over all locations in r from which $\jmath$ is reachable.

Expression dependence measures the syntactic similarity between predicates. Let $\mathsf{sub}(e)$ denote the set of sub-expressions of an expression e. The *expression proximity* $\mathsf{eprox}(e_1, e_2)$ of two expressions e_1, e_2 is the number of shared sub-expressions: $\mathsf{eprox}(e_1, e_2) \;=\; |\mathsf{sub}(e_1) \cap \mathsf{sub}(e_2)|$. Then, the *expression dependence* $\mathsf{edep}(p, c)$ is the normalized score measuring the syntactic similarity of p and c with respect to all expressions $\mathbb{P}_{r,c}$ mentioned in routine r and in c.

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

Dynamic Analysis. Dynamic analysis extracts evidence from test cases in the form of score associated to every state component generated. Given a component $\langle \ell, p, v \rangle$, the higher the score $\mathsf{dyn}\langle \ell, p, v \rangle$ it receives from dynamic analysis, the stronger the runtime behavior suggests that an error originates at location ℓ when predicate p evaluates to v.

Code-based fixing computes the dynamic score based on principles suggested by fault-localization techniques [7] and heuristically adjusted to the goal of program fixing (see [4]):

$$\mathsf{dyn}\langle \ell, p, v \rangle = \gamma + \frac{\alpha}{1-\alpha}\left(1 - \beta + \beta\alpha^{\#\mathsf{p}\langle \ell, p, v \rangle} - \alpha^{\#\mathsf{f}\langle \ell, p, v \rangle}\right)$$

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 exercise the component $\langle \ell, p, v \rangle$. Some empirical evaluation suggested to set $\alpha = 1/3$, $\beta = 2/3$, and $\gamma = 1$ in the current implementation of AutoFix-E2.

Combining Static and Dynamic Analysis. The final output of the analysis phase combines static and dynamic analysis to assign a "suspiciousness" score $\mathsf{fixme}\langle \ell, p, v \rangle$ to every state component $\langle \ell, p, v \rangle$ corresponding to a violation of contract clause c at location $\jmath$:

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

Fix Candidate Generation. Consider a component $\langle \ell, p, v \rangle$ with a high evidence score $\mathsf{fixme}\langle \ell, p, v \rangle$. $\langle \ell, p, v \rangle$ induces a number of possible *actions* (instructions) which try to avoid using the value v of p at ℓ. Such actions aim at sub-expressions of p that are modifiable at ℓ. Actions on expressions of reference type consists of routine calls with the expressions as target; actions on expressions of integer or boolean type consists of assignments of new values that may reverse common mistakes, such as "off-by-one" errors.

The fix actions can then be injected into the faulty code to form *candidate fixes* according to the fix schemas in Table I; for a state component $\langle \ell, p, v \rangle$, we instantiate each of the schemas in Table I as follows:

fail takes $p = v$, the component's predicate and value,
snippet takes any possible action,

(*a*)	(*b*)	(*c*)	(*d*)

snippet
old_stmt
	if *fail* **then**
	snippet
	end
	old_stmt
		if not *fail* **then**
		old_stmt
		end
			if *fail* **then**
			snippet
			else
			old_stmt
			end

TABLE I
FIX SCHEMAS.

old_stmt is the instruction at location ℓ.

Validation of Candidates. The validation phase runs each candidate fix through the full set of passing and failing test cases. A fix is *validated* if it passes all the previously failing test cases and it still passes the original passing test cases. In general, more than one candidate fix may pass the validation phase; AutoFix-E2 ranks all valid fixes according to the score of the state component that originated the fix and submits the top 15 to the user, who is ultimately responsible to decide whether to deploy any of them.

IV. EXPERIMENTAL EVALUATION

All the experiments ran on a Windows 7 machine with a 2.66 GHz Intel dual-core CPU and 4 GB of memory. On average, AutoFix-E2 ran for 7.6 minutes for each fault.

A. Selection of Faults

The experiments include faults from two sources: data structure classes from commercial libraries, and an implementation of a library to manipulate text documents developed as student project.

Data structure libraries. Table II lists the 15 classes from the EiffelBase [8] and Gobo [9] libraries used in the experiments; the table reports the length in lines of code (LOC), the total number of routines (#R) and boolean queries (#B) of each class, and the number of faults (#F) considered in the experiments. This selection of faults combines 13 faults used in the evaluation of model-based fixing [3] with 51 new faults recently found by AutoTest. We did not re-use the remaining 29 faults used in [3] because they are not reproducible in the latest revision of the libraries.

A library to manipulate text documents. The second part of the evaluation targets a library to manipulate text documents. The library was implemented as a student project at ETH Zurich in 2010. Table III lists the 3 classes of the library used in the experiments, with the same statistics as in Table II. Compared to EiffelBase and Gobo, the text document library's classes have a more primitive interface, with very few boolean queries (31 of the 32 boolean queries of class *FILE_NAME* are inherited from the library class *STRING*, hence they are mostly unrelated to the specific semantics of *FILE_NAME*) and less detailed contracts; therefore, they are representative of less mature software with functionalities complex to specify formally. AutoTest detected 9 faults (#F) in the classes.

TABLE II
EIFFELBASE AND GOBO CLASSES.

CLASS	LOC	#R	#B	#F
ACTIVE_LIST	2162	139	19	2
ARRAY	1464	101	11	9
ARRAYED_CIRCULAR	1910	133	25	3
ARRAYED_SET	2345	146	18	5
DS_ARRAYED_LIST	2762	166	9	5
DS_HASH_SET	3076	169	10	1
DS_LINKED_LIST	3434	160	8	5
HASH_TABLE	2036	118	19	2
INTEGER_32	1115	99	5	1
LINKED_LIST	2000	109	16	1
LINKED_PRIORITY_QUEUE	2374	125	17	1
LINKED_SET	2352	122	16	5
REAL_64	839	72	4	1
SUBSET_STRATEGY_HASHABLE	543	33	0	4
TWO_WAY_SORTED_SET	2868	141	18	19
Total	31280	1833	195	64

TABLE III
DOCUMENT MANIPULATION LIBRARY CLASSES.

CLASS	LOC	#R	#B	#F
FILE_NAME	4297	258	32	2
HTML_TRANSLATOR	1148	83	0	1
LATEX_TRANSLATOR	1269	90	0	6
Total	6714	431	32	9

Selection of Test Cases. All the experiments used test cases generated automatically by AutoTest; this demonstrates complete automation of the whole debugging process and minimizes the potential bias introduced by experimenters. AutoTest produced an average of 25 passing and 11 failing test cases for each fault.

B. Experimental Results

Data Structure Libraries. Table IV summarizes the results of the experiments on the data structure libraries: the number of faults in each category, the faults fixed with model-based techniques using AutoFix-E, and those fixed with code-based techniques using AutoFix-E2. The count of valid fixes only includes those which are *proper*, i.e. which manual inspection confirmed to be adequate beyond the correctness criterion provided by the contracts and tests available.

The results show that code-based techniques constitute a significant improvement over model-based techniques. The faults fixed by AutoFix-E2 are a superset of those fixed by AutoFix-E: when both tools succeeded, they produced equivalent fixes (with possibly negligible syntactic differences); most of the errors where code-based fixing succeeds and model-based techniques fail are indicative of subtle bugs with non-obvious fixes. The 50 faults not fixed are, to our knowledge, "deep" errors beyond the capabilities of any existing automatic program fixing technique.

Text Document Manipulation Library. The second set of experiments tried to determine if code-based techniques can successfully tackle software beyond well-engineered data structure implementations. AutoFix-E2 built valid fixes for 5 of the 9 faults in the document library. In comparison, AutoFix-E only fixed one of the faults, which AutoFix-E2 also fixed; manual inspection confirmed the expectation that model-based fixing fails whenever the fault conditions cannot be characterized using only boolean queries.

TABLE IV
FAULTS FIXED IN EIFFELBASE AND GOBO CLASSES.

TYPE OF FAULT	# F	MODEL	CODE
Precondition violation	22	10 (45%)	12 (54%)
Postcondition violation	30	0 (0%)	2 (6%)
Call on void target	7	0 (0%)	0 (0%)
Intermediate assertion violation	5	0 (0%)	0 (0%)
Total	**64**	**10(15%)**	**14(22%)**

Overall Performance of Code-Based Fixing.

In the experiments, code-based techniques fixed 19 errors, 73% more than model-based techniques.

V. CONCLUSION

This paper introduced code-based automated program fixing, a novel approach to generate automatically corrections of errors in software equipped with contracts. Preliminary experiments with the supporting tool AutoFix-E2 demonstrate that code-based techniques extend the applicability of automated program fixing to more faults in classes beyond well-designed data structure implementations. An extended version of this paper, including related work, is available as technical report:

http://arxig.org/abs/1102.1059

Availability. The AutoFix-E2 source code, and all data and results cited in this article, are available at:

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

Acknowledgments. This work is partially funded by Hasler-Stiftung Grant no. 2327 "AutoFix—Programs that fix themselves". The facilities provided by the Swiss National Supercomputing Centre (CSCS) ran AutoTest to generate some of the test cases used in the experiments. The authors thank Andreas Zeller for the ongoing collaboration on automated program fixing. Bernhard Brodowsky, Severin Heiniger, and Stefan Heule implemented the text document library.

REFERENCES

[1] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *ASE*. ACM, 2007, pp. 433–436.
[2] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *ASE*. IEEE Computer Society, 2009, pp. 550–554.
[3] 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.
[4] Y. Pei, Y. Wei, C. A. Furia, M. Nordio, and B. Meyer, "Code-based automated program fixing," http://arxiv.org/abs/1102.1059, 2011.
[5] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf, "Programs that test themselves," *Computer*, vol. 42, no. 9, pp. 46–55, 2009.
[6] B. Meyer, *Object-Oriented Software Construction*. Prentice Hall, 1997.
[7] W. E. Wong, V. Debroy, and B. Choi, "A family of code coverage-based heuristics for effective fault localization," *J. Syst. Softw.*, vol. 83, no. 2, pp. 188–208, 2010.
[8] http://freeelks.svn.sourceforge.net.
[9] http://sourceforge.net/projects/gobo-eiffel/.