Execution-free program repair

Li Huang
Bertrand Meyer
Ilgiz Mustafin
Manuel Oriol
Li.Huang@constructor.org
Bertrand.Meyer@inf.ethz.ch
Ilgiz.Mustafin@constructor.org
mo@constructor.org
Constructor Institute of Technology / Constructor University
Schaffhausen, Switzerland

ABSTRACT

[Preprint. Appeared in FSE 2024, Foundations of Software Engineering, July 2024.]

Automatic program repair usually relies heavily on test cases for both bug identification and fix validation. The issue is that writing test cases is tedious, running them takes much time, and validating a fix through tests does not guarantee its correctness. The novel idea in the Proof2Fix methodology and tool presented here is to rely instead on a program prover, without the need to run tests or to run the program at all. Results show that Proof2Fix automatically finds and fixes significant historical bugs.

CCS CONCEPTS

• **Software and its engineering → Formal software verification**; **Software testing and debugging**; **Empirical software validation**; **Error handling and recovery**.

KEYWORDS

Program proofs, Tests and Proofs, SMT-solvers, Eiffel, Counter-Example

ACM Reference Format:

Li Huang, Bertrand Meyer, Ilgiz Mustafin, and Manuel Oriol. 2024. Execution-free program repair. In *Proceedings of the 32nd ACM Symposium on the Foundations of Software Engineering (FSE '24), November 15–19, 2024, Porto de Galinhas, Brazil*. ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 OVERVIEW

Finding bugs is good; correcting them is better. An important trend in the recent evolution of software verification research is the development of methods of Automatic Program Repair (APR), which propose corrections to bugs. Whereas results are promising, most of the existing APR approaches are *dynamic* [10, 11]: they infer and validate the corrections by running the program. The present work describes a static, *prover-based* approach, Proof2Fix, which only needs the source code to find and fix bugs.

Automatic program repair typically involves four steps [7]: identify a fault (also called a "bug"); localize the fault; generate a fix (also called "correction" or "patch"); evaluate the fix. Moving from a dynamic test-based approach to a static prover-based approach benefits all these goals but particularly two:

- *Fault identification*: A dynamic approach needs to prepare test inputs, often many of them, and run tests until one triggers the bug. In spite of progress towards automatic techniques [15, 18, 21], test case preparation remains, in most practical cases, a considerable labor-intensive task. With the static approach presented here, one simply runs a prover on the program text; bugs are detected when the prover fails to show that the program is correct. [1] No need to invent test data or produce a test harness.
- *Fix validation*: this step is perhaps where a prover-based approach has the most substantial advantage. Once we have a candidate fix, a dynamic approach must run all the tests again and check that they pass; not only is this process time-consuming, it is also fraught with uncertainty since a positive answer is only as good as the test suite, by nature non-exhaustive. In a prover-based approach the fix validation is a proof, which provides a guarantee that the fix is correct (or not).

A prover-based approach can also help *Fault localization*, by pinpointing (often down to the level of a particular instruction) the precise program element that precludes verification, and *Fix generation*, by avoiding biases.

Taking advantage of a modern proof environment, the Proof2Fix approach and tool described in the following sections produce meaningful fixes and validates them formally.

2 A SAMPLE SESSION

To explain the main idea before going into the underlying technology, we start with a typical use of the Proof2Fix on a small but representative example (Fig. 1). The `CLOCK` class, in Eiffel, implements a digital clock with fields `seconds`, `minutes` and `hours`, and routines to

FSE'24, July 2024, Porto de Galinhas, Brazil, Brazil

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

[1] Note that the limitation of the prover sometimes prevents it to assess code correctness. With modern provers this case can largely be eliminated or worked around.

```
1  class CLOCK feature
2  hours: INTEGER -- Hours of clock.
3  minutes: INTEGER -- Minutes of clock.
4  seconds: INTEGER -- Seconds of clock.
5  increase_hours -- Go to next hour.
6      do
7          if hours = 24 then hours := 0
8          else hours := hours + 1 end
9      ensure hours_increased: hours = (old hours + 1) \\ 24
10     end
11 increase_minutes -- Go to next minute.
12     do
13         if minutes <59 then minutes := minutes + 1
14         else minutes := 0 end
15     ensure
16     hours_increased: old minutes= 59 ⟹ hours=(old hours+1)\\24
17     hours_unchanged: old minutes<59 ⟹ hours = old hours
18     minutes_increased: minutes = (old minutes + 1) \\ 60
19     end
20 invariant
21     hours_valid: 0 ≤ hours ∧ hours ≤ 23
22     minutes_valid: 0 ≤ minutes ∧ minutes ≤ 59
23     seconds_valid: 0 ≤ seconds ∧ seconds ≤ 59
24 end
```

Figure 1: A buggy version of the CLOCK class

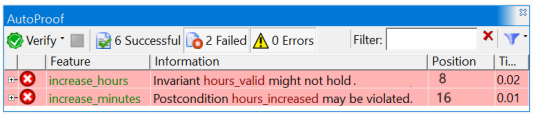

Figure 2: AutoProof report of proof failures for class CLOCK

increment their values: `increase_hours` and `increase_minutes`. Contracts express the semantics: preconditions (**`require`**), postconditions (**`ensure`**), and class **`invariant`**. The postcondition of `increase_hours` (line 9) specifies the relation between the values of `hours` on routine entry (denoted **`old`** `hours`) and exit; the invariant (21 – 23) states the values' validity ranges. The task of the prover, here AutoProof [19] — a prover for contract-equipped program, part of a technology stack that includes the Boogie [6] proof engine and the Z3 SMT solver [3] — is to establish that every routine execution starting in a state satisfying the precondition and invariant terminates in a state satisfying the postcondition and again the invariant.

The proof attempt with AutoProof fails, producing the messages of Fig. 2. The first failure results from using 24 instead of 23 as the threshold in line 7; the second one, from forgetting to increase `hours` when `minutes` reaches 59 (line 14). Proof2Fix automatically generates fixes for both bugs, as shown in Fig. 3. The fix for `increase_hours` properly identifies the faulty case for `hours = 23`, and replaces the **`if`** condition accordingly; the fix for `increase_minutes` correctly calls `increase_hours` when `minutes = 59`. With these two fixes, AutoProof now succeeds in verifying the routines.

3 EXECUTION-FREE FIX GENERATION

Fig. 4 summarizes the Proof2Fix process. AutoProof attempts to verify the class. If it discovers a fault in the program, it reports a proof failure and produces, by leveraging the underlying SMT

```
increase_hours_fixed
do
-       if hours = 24 then
+       if hours = 23 then
            hours := 0
        else
            hours := hours + 1
        end
end
```

```
increase_minutes_fixed
do
    if minutes < 59 then
        minutes := minutes + 1
    else
        minutes := 0
+       increase_hours
    end
end
```

Figure 3: Generated fixes (highlighted)

solver (such as Z3), a set of counterexamples (CEs)[2] illustrating the failure. Each contains a trace (a sequence of states) documenting how the program ends up in a state that violates a contract. It then creates *CE invariants*: predicates that always hold in the CEs. Proof2Fix derives them from *state invariants* — predicates that always hold at a certain state in a set of program executions — which it infers using Daikon [4].

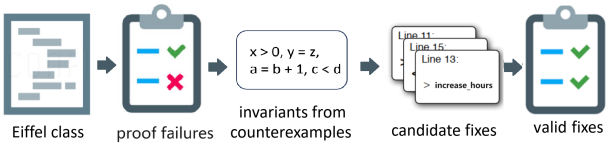

Figure 4: Proof2Fix workflow

The resulting CE invariants[3] characterize the circumstances for a failure to happen; they guide Proof2Fix's process when generating candidates fixes. Proof2Fix then runs AutoProof again to retain only the fixes that both remove the original proof failure and introduce no extra failure. The rest of section 3 details these two key steps: inferring invariants from CEs (3.1) and synthesizing candidate fixes based on the invariants (3.2).

3.1 Inferring CE invariants

For a class C and one of its routines r, let $c_1, ..., c_n$ be the set of CEs produced by the prover when a failure of r occurs. Each c_i contains a sequence of states, represented by $s_1^{c_i}, ..., s_m^{c_i}$. Each such state $s_j^{c_i}$ $(1 \le j \le m)$ consists of a set of equalities between program expressions — including C's attributes (fields) and r's arguments — and their values in that state.

To derive invariants that hold in all CEs, Proof2Fix feeds the CEs into Daikon, which yields m sets $inv_1, ..., inv_m$, where each inv_j is a set of candidate invariants which hold in the j^{th} state in every CE. Formally:

$$\forall c_i \in \{c_1, ..., c_n\}, s_j^{c_i} \models inv_j$$

The derivation relies on a set of predefined templates including:

- Equality to primitive-type constants: `e = v` where a boolean or integer expression `e` has the value `v`. In Fig. 3, we get `hours = 23` and `minutes = 59`.
- Non-constant equality: `e1 = e2` where `e1` and `e2` (not both constants) have the same value.
- Linear relations: properties of the form `e1 = a·e2 + b` or `e1 = a·old e2 + b`, for integer expressions `e1`, `e2` and constants `a`, `b`.

[2] By running the verification multiple times with different seeds.
[3] "Invariants" as used in Daikon and the present discussion are more general than the OO concept of class invariant (section 2).

The set of invariants obtained from these templates, denoted P, is the *basic invariants set*; Proof2Fix derives a *compound invariants set* Π by combining pairs of basic invariants in P through disjunctions. For example, when P = {hours = 23, minutes≥0}, then Π = {hours = 23, minutes≥0, hours = 23 ∨ minutes≥0 }.

The current implementation of Proof2Fix only takes advantages of the invariants in the *initial state*, characterizing a set of faulty inputs that leads to the proof failure and appear to be the most interesting. Future work will explore invariants of other states.

3.2 Synthesize candidate fixes

Having obtained CE invariants (elements ϕ of the set Π) of a routine r, Proof2Fix can now produce candidate fixes of two kinds, described next: *contract* fixes, which affect the specification of r (contract), and *implementation* fixes, which modify its implementation (body).

It is generally beyond Proof2Fix's purview to determine which one, of contract and body, should be fixed; if it finds fixes of both kinds, it will let the programmer decide. Empirical studies [9] indeed indicate that in practice bugs arise from both kinds of mistake.

Contract fixes use one of the following strategies:

- **Precondition strengthening**: add "`not` ϕ" to r's precondition, to rule out the faulty cases characterized by ϕ.
- **Postcondition weakening**: let ψ be the postcondition or class invariant that fails in the proof, Proof2Fix replaces it with a weakened version "`not` ϕ `implies` ψ", making the previously failing cases now verify.

Precondition strengthening requires special care. This strategy is often useful, since forgetting a precondition clause is a common source of bugs (as in forgetting to state that the argument of a real-square-root routine must be non-negative). Applied without restraint, however, it could lead to absurdities; in fact, any routine will verify if we preface it by `require False`. Some cases are more subtle: a proposed fix might inadvertently add a clause `not` ϕ contradicting existing clauses (as in a supposed fix that adds the precondition clause a > 0 whereas r's precondition includes a = 0). To filter out such spurious fixes, Proof2Fix deliberately injects a fault into r by inserting a contradictory assertion (`check False end`[4]) at the beginning of its body; any candidate fix whose verification does not capture the injected fault will be discarded.

Generating **Implementation fixes** involves two steps: selecting a fix schema that abstracts common instruction patterns; instantiating the fix schema with proper instructions and ϕ.

Fig. 5 shows the currently implemented fix schemas. `snippet` is a sequence of instructions, and `old_stmt` represents some instructions in the original program related to the point of failure. Proof2Fix assumes that the variables used in the violated postcondition are suspicious; when instantiating `snippet`, it selects instructions i that alter the states of the variables, based on their types:

- Boolean. i is an assignment e := d where d is the constant `True` or `False` or is the expression `not` e.
- Integer. i is an assignment e := d where d is one of to the constants 0, 1, and −1, or the expressions −e, e + 1, e − 1.
- Reference. i is a call to a command (procedure) e.p (a_1,..., a_n), where p is a routine available to the faulty routine and a_1,..., a_n are the arguments.

`old_stmt` is one of the following:

- The single instruction l at the failure's location.
- The block of instructions that immediately contains l.

Accordingly, the instantiated schema replaces the instruction at the failure's location or the whole block. The two fixes in Fig. 3 are generated by instantiating the fix schemas (a) and (d).

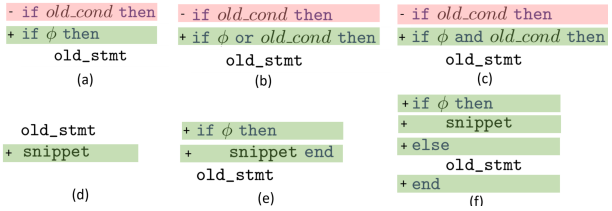

Figure 5: Fix schemas of Proof2Fix

To improve performance when r contains multiple blocks, Proof2Fix identifies those involved in all CEs and treats them in priority, as the ones most likely to be faulty.

4 EVALUATION

This section reports a preliminary assessment of the quality of the automatically generated fixes from Proof2Fix on a collection of faulty Eiffel programs[5], including data-structure classes adapted from an old version of EiffelBase[6], examples in the AutoProof tutorial[7], benchmarks of previous software verification competitions [1, 5], and a benchmark of recursive programs[8]. The experiment ran on a Windows 11 machine with a 2.10 GHz Intel 12-Core processor and 32 GB of RAM. The number of CEs used in invariant inference for each failure is set to 10 empirically. The EiffelBase examples (started with "V_" in Table 1) are particularly significant as they are real bugs (not seeded), which actually arose in earlier released versions of the library, and were subsequently corrected[9].

Table 1 lists, for each class, its length in lines of code (LOC), the number of failures (#Failures) detected by AutoProof, and for how many of those failures Proof2Fix built (at least one) valid (#Valid) or proper (#Proper) fixes, as well as the average fix generation time (in seconds) for each failure (Avg.#T_f).

The difference between the last two categories comes from the usefulness (in the current state of the approach) of a human reasonableness check. A fix is valid if the repaired program passes verification. Some valid fixes, however, might be over-enthusiastically change the intended semantics of the program; the most important case is the weakening of a postcondition, which is sometimes justified but sometimes results in a useless program. For that reason, we currently perform a manual check of valid fixes and discard any that would distort the intuitively understood intent of the code. *Proper* fixes are valid fixes that pass this human check. Valid but

[4] `check` p `end`, equivalent to `assert` p in some other formalisms, has no effect on execution but verifies only if p holds at the given program point.

[5] All code and results are available at https://github.com/proof2fix/proof2fix.
[6] https://www.eiffel.org/doc/solutions/EiffelBase
[7] http://AutoProof.sit.org/AutoProof/tutorial
[8] https://github.com/maple-repair/recursive-benchmark
[9] Since the technology has evolved, some of the examples would no longer compile in their original form; we re-injected the original bugs into the current versions.

Table 1: Fixing Results of Proof2Fix

Classes	LOC	#Failures	#Valid	#Proper	Avg.T_f (s)
ABSOLUTE	18	2	2	2	49
CONSEQ	17	4	4	3	55
INCREMENT	17	4	4	2	49
MAX	17	4	4	1	71
MIN	26	4	4	1	64
SUM	20	8	4	2	48
ACCOUNT	102	7	5	2	54
CLOCK	141	9	9	5	89
ARITHMETIC	190	4	3	2	50
HEATER	78	4	3	3	169
LAMP	81	4	4	0	147
J_ABSOLUTE	41	12	12	5	51
MAX_IN_ARRAY	40	5	0	0	58
SQUARE_ROOT	45	4	3	0	47
V_ARRAY	67	1	1	1	61
V_ARRAYED_LIST	76	1	0	0	91
V_INDEXABLE_CURSOR	123	1	1	0	98
V_LINKED_LIST	38	2	2	1	78
Total	1137	80	66	30	70

improper fixes include: too-restrictive precondition strengthening; too-generous postcondition strengthening (as already mentioned), including an implication "`not` $\phi \implies \psi$" where ϕ in fact always evaluates to `true`; a spurious fix that recursively calls the routine itself to ensure its postcondition. Note that improper valid fixes remain valuable, not as fixes but as debugging aids, since they clearly evidence failure-inducing inputs.

Among the 80 failures detected by AutoProof, Proof2Fix is able to generate at least one valid fix for 82.5% of them and at least one proper fix for 37.5% of the failures. On average, Proof2Fix ran for 70 seconds for each failure, which seems an acceptable overhead. For each of the 66 fixed failures, Proof2Fix generated, on average, 124 candidates and 5 valid fixes; the average percentage of valid fixes per candidate is 4%. Fig.6 shows the correlation between the fix time and the number of candidate fixes (#Cand); the increment of #Cand would not lead to significant increase of time: when #Cand < 100, time increases very slowly; when #Cand is beyond 100, time roughly grows linearly.

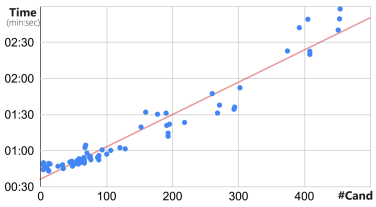

Figure 6: Fix time over number #Cand of candidate fixes

Table 2 correlates the causes of proof failures with the effectiveness of Proof2Fix in finding valid fixes for that category of failures.

Table 2: Failure causes and fixes

Cause of Failure	#Failure	#Fixed	#Proper Fixed
incorrect expression of an assignment	25	21 (84%)	12 (48%)
incorrect condition of an if statement	18	17 (94%)	12 (67%)
weakness of assisting contracts (postcondition of a callee or loop invariant)	10	3 (30%)	0 (0%)
missing instruction (assignment or routine call)	7	7 (100%)	3 (43%)
missing precondition	6	5 (83%)	2 (33%)
incorrect postcondition	4	4 (100%)	0 (0%)
incorrect algorithm	7	6 (100%)	1 (14%)
others	3	3 (100%)	0 (0%)

From these results, Proof2Fix is most effective for failures caused by incorrect expression of an assignment and incorrect condition of an `if` statement. In both cases, Proof2Fix can obtain CE invariants that accurately characterize the faulty cases that need to be ruled out; Proof2Fix is not good at fixing failures caused by weakness of assisting contracts, as the CEs generated in those cases usually vary a lot, making it difficult to infer useful invariants.

5 PREVIOUS WORK

AutoFix [16, 17] performs automated fixing of programs based on patterns for fixing the program. The main differences with Proof-2Fix are that it locates faults by using dynamic program analysis (which is more time consuming) instead of proofs, and it proposes fixes based on rather simple heuristics and verifies them using a test suite (instead of proving the resulting code). A tool based on similar ideas is JAID [2], which first extracts the contracts, then applies fixing and validation techniques similar to those of AutoFix. Other tools such as SemFix [12] or Nopol [22] also use test suites as a verification tool for fixes generated through symbolic execution. Nilizadeh [14] showed that this is prone to overfitting. In contrast, Proof2Fix's fixes are guaranteed to be correct, a result that no test-based approach can provide. In addition, the prover-based approach provides precise location identification for the failure.

Closer to Proof2Fix are FootPatch [20], Maple [13], and the approach of Logozzo and Ball [8], which use a formal approach for identifying faults, creating and verifying fixes. These three tools address and fix specific classes of faults: memory errors for (FootPatch); issues that can be translated to linear arithmetic expressions (Maple); contracts, initialization, method purity and guards (Logozzo and Ball). Proof2Fix is more generic and has the potential to fix any fault discovered by the prover and verify the fixes.

6 PLANNED DEVELOPMENTS

Our goal is to make Proof2Fix part of a standard development environment such as EiffelStudio (the main Eiffel IDE). Although the experiment results have demonstrated the effectiveness of the Proof2Fix in fixing bugs on various examples, including (most convincingly in our view) historical bugs in production code, several further developments are necessary for the approach to deliver its full value.

Optimization of CE invariants. The quality of generated fixes from Proof2Fix relies on the quality of CE invariants. We are working on more sophisticated kinds of invariants, including first-order predicates with quantifiers. We are also improving the predicate sets by spotting and removing redundancies and contradictions.

Fix schema generality. The current implementation does not instantiate fix schemas with more than one predicate at a time and empirically limits the maximum admitted length of a snippet to a small number. We are developing more sophisticated techniques.

Bug and fix diversity. More generally, a major goal of our current extension work is to expand the range of bug and fix types dramatically, by performing an extensive review of historical bugs and corrections in large software repositories and deriving a set of patterns covering as many of them as possible, based on the conjecture that while the set of real-world bugs and fixes is large, many of them can be attached to a tractable number of categories.

Fix ranking. Proof2Fix often finds several valid fixes for a given failure. We are working towards ranking them properly.
Number of CEs The maximum number of CEs is, so far, fixed. We are devising rigorous, empirically validated criteria for choosing the best value for every case.
IDE integration With the promise of tools such as Proof2Fix, Automatic Program Repair can become, rather than an esoteric facility to be run on demand, a component of the normal practice of developing, verifying and debugging programs. We are working towards including Proof2Fix as an integral part of a production-grade software development environment.

7 CONCLUSIONS

Programmers deserve Automatic Program Repair: mechanisms to suggest correct, trustworthy and reliable fixes to the mistakes they inevitably made, and to mistakes inevitably made by any AI-based tools that become part of their tool base. Using a completely static approach, based on a powerful state-of-the-art program-proving tool stack, overcomes the inherent limitations of dynamic (test-based approaches): it is faster, does not require preparing test cases, works on simulated environments (particularly for embedded and cyberphysical systems), and — perhaps even more importantly — guarantees that proposed fixes are valid corrections. We believe that Proof2Fix, as presented in this article, provides a promising step towards making Automatic Program Repair a realistic everyday component of modern software development.

REFERENCES

[1] Thorsten Bormer, Marc Brockschmidt, Dino Distefano, et al. 2011. The COST IC0701 Verification Competition. In *International Conference on Formal Verification of Object-Oriented Software (FoVeOO)*. Springer, 3–21.
[2] Liushan Chen, Yu Pei, and Carlo A Furia. 2017. Contract-based program repair without the contracts. In *International Conference on Automated Software Engineering (ASE)*. IEEE, 637–647.
[3] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 337–340.
[4] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming* 69, 1-3 (2007), 35–45.
[5] Vladimir Klebanov, Peter Müller, , et al. 2011. The 1st Verified Software Competition: Experience Report. In *International Symposium on Formal Methods (FM)*. Springer, 154–168.
[6] Claire Le Goues, K Rustan M Leino, and Michał Moskal. 2011. The Boogie Verification Debugger. In *International Conference on Software Engineering and Formal Methods (SEFM)*. Springer, 407–414.
[7] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Transactions on Software Engineering* 38, 1 (2011), 54–72.
[8] Francesco Logozzo and Thomas Ball. 2012. Modular and Verified Automatic Program Repair. *SIGPLAN Notices* 47, 10 (2012), 133–146.
[9] Bertrand Meyer, Ilinca Ciupa, Lisa Ling Liu, Manuel Oriol, Andreas Leitner, and Raluca Borca-Muresan. 2007. Systematic evaluation of test failure results. In *Workshop on Reliability Analysis of System Failure Data (RAF)*.
[10] Martin Monperrus. 2018. Automatic software repair: A bibliography. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–24.
[11] Martin Monperrus. 2018. *The living review on automated program repair*. Ph. D. Dissertation. HAL Archives Ouvertes.
[12] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *International Conference on Software Engineering (ICSE)*. IEEE, 772–781.
[13] Thanh-Toan Nguyen, Quang-Trung Ta, and Wei-Ngan Chin. 2019. Automatic program repair using formal verification and expression templates. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Springer, 70–91.
[14] A. Nilizadeh, G. T. Leavens, X. D. Le, C. S. Pasareanu, and D. R. Cok. 2021. Exploring true test overfitting in dynamic automated program repair using formal methods. In *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 229–240.
[15] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 815–816.
[16] Yu Pei, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. 2015. Automated Program Repair in an Integrated Development Environment. In *International Conference on Software Engineering (ICSE)*, Vol. 2. IEEE, 681–684.
[17] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. 2014. Automated Fixing of Programs with Contracts. *Transactions on Software Engineering* 40, 5 (2014), 427–449.
[18] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex–White Box Test Generation for .Net. In *International Conference on Tests and Proofs (TAP)*. Springer, 134–153.
[19] Julian Tschannen, Carlo A Furia, Martin Nordio, and Nadia Polikarpova. 2015. Autoproof: Auto-active Functional Verification of Object-Oriented Programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 566–580.
[20] Rijnard van Tonder and Claire Le Goues. 2018. Static Automated Program Repair for Heap Properties. In *International Conference on Software Engineering (ICSE)*. ACM, 151–162.
[21] Yi Wei, Serge Gebhardt, Bertrand Meyer, and Manuel Oriol. 2010. Satisfying Test Preconditions Through Guided Object Selection. In *International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 303–312.
[22] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clément, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *Transactions on Software Engineering* 43, 1 (2017), 34–55.