Invariant-based Program Repair

Omar I. Al-Bataineh(✉)

Simula Research Laboratory, Oslo, Norway
omar@simula.no

Abstract. This paper describes a formal general-purpose automated program repair (APR) framework based on the concept of program invariants. In the presented repair framework, the execution traces of a defected program are dynamically analyzed to infer specifications $\varphi_{correct}$ and $\varphi_{violated}$, where $\varphi_{correct}$ represents the set of likely invariants (good patterns) required for a run to be successful and $\varphi_{violated}$ represents the set of likely suspicious invariants (bad patterns) that result in the bug in the defected program. These specifications are then refined using rigorous program analysis techniques, which are also used to drive the repair process towards feasible patches and assess the correctness of generated patches. We demonstrate the usefulness of leveraging invariants in APR by developing an invariant-based repair system for performance bugs. The initial analysis shows the effectiveness of invariant-based APR in handling performance bugs by producing patches that ensure program's efficiency increase without adversely impacting its functionality.

Keywords: Automated program repair · Invariant learning and refinement · Patch overfitting · Program verifier · CPAChecker · Performance bugs

1 Introduction

Automated program repair (APR) has recently gained great attention because it helps to significantly decrease manual debugging effort by automatically generating patches for defected programs. Modern program repair tools have been shown to be effective at fixing bugs in many real-world programs. The poor quality of automatically generated patches [11], however, continues to be a major obstacle to the adoption of automated program repair by software practitioners.

Problem: The primary reason for the low quality of automatically generated patches by current APR tools is the lack of specifications of the intended behavior. Most program repair systems rely on tests as the correctness criteria, because a formal specification is not explicitly provided by software developers. Therefore, current APR approaches produce plausible patches which must be (manually) inspected before being deployed. As a result, there is no guarantee that the generated patches are generally correct and do not introduce new bugs.

Solution: Program verification technology enables developers to prove the correctness of the program before deploying it. One of the key activities underlying this technology involves inferring a program invariant—a logical formula that

This work is supported by the Research Council of Norway through the secureIT project (IKTPLUSS #288787).

D. Beyer and A. Cavalcanti (Eds.): FASE 2024, LNCS 14573, pp. 255–265, 2024.
https://doi.org/10.1007/978-3-031-57259-3_12

serves as an abstract specification of a program. Developers can significantly benefit from program invariants to identify program properties that must be preserved when modifying code. Unfortunately, these invariants are typically absent from code, leading to the dominance of less rigorous APR approaches (e.g., dynamic APR) and the well-known patch overfitting challenge [11].

We argue that by using test cases and reachability-based analysis techniques, an accurate set of invariants may be obtained and utilized to produce high-quality patches. In other words, program verification tools such as CPAChecker [3] and PathFinder [15] can be used to refine the dynamically generated invariant candidates. This can be done by first using the test cases to analyze the execution traces of the program to infer a set of invariant candidates. These candidates are then refined using a program verifier to obtain more accurate invariants. The goal is to infer two specifications: (i) $\varphi_{correct}$, which represents the set of *good patterns* required for a run to succeed, and (ii) $\varphi_{violated}$, which represents the set of *bad patterns* that lead to the target bug. Invariant-based APR offers two key benefits. First, it directs APR towards potentially feasible patches. Second, it enables the formal validation of plausible patches using program verifiers.

Viability of invariant-based APR: Program invariants have shown effectiveness in many applications, such as program understanding, fault localization, and formal verification. Invariants are effective because functional correctness relates to the final result of a program rather than any specific implementation. They can therefore assist in abstracting many concrete execution steps and thus greatly reduce the effort needed to reason about the patch's correctness.

In fact, developers who aim to repair a defected *undocumented program* (a program written without thought for formal specifications) can find invariant-based APR helpful in their repair tasks. The availability of mature automated invariant detection tools like Daikon [4] and practical software verification tools like CPAChecker and PathFinder makes the invariant-based program repair technique viable. At first glance, refining invariants using program verification tools seems too expensive. However, due to tremendous advances in software verification [2], in practice, invariant-based verification can be made pretty efficient. In particular, the software analysis framework CPAChecker, which supports many different reachability analyses, has been effectively used to validate a wide variety of reachability queries against C programs with up to 50K lines of code. This makes reachability analysis a promising technique that can be used to significantly reduce the patch overfitting problem and produce high-quality patches.

2 Invariant-based Program Repair Framework

In this section we reformulate the APR problem using the concept of program invariants. We then describe how one can analyze the execution traces of fault-free runs to infer likely specifications of the program's intended behaviour and execution traces of faulty runs to infer likely suspicious invariants that lead to the faulty behaviour. Before proceeding further, let us introduce some definitions.

Definition 1. *(**fault-free vs. faulty runs**). Let P be a buggy program, $\mathcal{R}$ be the set of runs of P, and φ_{beh} be a property of program P's intended behavior. We say that a run $r \in \mathcal{R}$ is a successful run (i.e., fault-free run) if $P(r) \models \varphi_{beh}$. On the other hand, we say that a run $r' \in \mathcal{R}$ is a faulty run if $P(r) \not\models \varphi_{beh}$.*

From Definition 1 we note that by analyzing information extracted from fault-free runs, one might be able to infer a specification of the program's intended behavior. Similarly, by analyzing the execution information of faulty runs, one might be able to deduce the violating invariants that cause the bug. This is because fault-free runs represent runs in which program invariants are maintained, while faulty runs represent runs in which some program invariants are violated.

Definition 2. *(**Invariant-based APR problem**). Let P be a program containing bug b and $T = (T_P \cup T_F)$ be a test suite, where T_P represents the set of passing tests and T_F represents the set of failing tests. Let D be a dynamic invariant inference tool like Daikon, and V be a program verification tool like CPAChecker. The invariant-based APR process consists of the following steps:*

1. [*Invariant extraction*]. *Generate an initial set of invariants $\mathcal{I}$ for P using D.*
2. [*Invariant refinement*]. *Refine the set $\mathcal{I}$ using V to produce specifications $\varphi_{correct}$ and $\varphi_{violated}$. This can be done by asserting invariants at a program's location of interest and using any generated counter-example to refine them.*
3. [*Fault localization*]. *Compute a list of suspicious statements whose mutation may lead to a valid patch by analyzing specifications $\varphi_{correct}$ and $\varphi_{violated}$.*
4. [*Patch generation*]. *Construct code that corrects the invariants that are violated while maintaining other program invariants. This can be performed by employing a patch generation procedure like search- or semantic-based.*
5. [*Patch validation*]. *Validate the correctness of the generated patches using V.*

Depending on the type of the bug being fixed and the structure of the analyzed program, different program locations may be of relevance for properties $\varphi_{correct}$ and $\varphi_{violated}$. Examples include pre- and post-conditions for different functions, or loop invariants for some program loops. Note that the first two steps of the invariant-based APR process described at Definition 2 are necessary for increasing confidence in the precision of patches that are generated. The actual repair steps of the process, steps 3-5, can be formally stated as follows:

$$pt = FV(PGV(FL(\varphi_{correct}, \varphi_{violated}, P), T), \varphi_{correct}, \varphi_{violated}) \tag{1}$$

where FL is an invariant-based fault localization process, PGV is patch generation and validation process using test suite, and FV is a formal patch validation process using the verification tool V. If no plausible patch is found or a plausible patch is found but incorrect, the repair process returns fail. However, if the plausible patch passes the verification step carried out by the tool V, the process returns a patch. We now turn to discuss how one can generate specifications $\varphi_{correct}$ and $\varphi_{violated}$ by analyzing the execution information obtained by running program P using passing and failing tests. The analysis of fault-free and faulty runs leads to the identification of the following formal patterns.

1. $\varphi_{correct} = \mathcal{I}_{good} = V(D(P, T_P))$, invariants deduced using only successful runs. This set of invariants represents the likely intended behavior of P.
2. $\varphi_{faulty} = \mathcal{I}_{mix} = V(D(P, T_F))$, invariants deduced using the set of faulty runs. Note that the set $\mathcal{I}_{mix}$ may contain both good and bad patterns depending on how the target bug affects different functionalities of P.
3. $\varphi_{violated} = (\mathcal{I}_{mix} \setminus \mathcal{I}_{good})$, the set of violated invariants related to the bug.

It is important to categorize and distinguish inferred patterns (invariants) into good and bad patterns, especially when dealing with programs that have several functional requirements. This helps to identify the set of desired invariants to be maintained and violated invariants to be repaired when modifying code. It also helps to identify the set of invariants that are relevant to the analyzed bug. The soundness of inferred $\varphi_{correct}$ and $\varphi_{violated}$ depends heavily on the soundness of the employed invariant inference tool as well as the invariant refinement process. Increasing the amount of program behavior exercised using reachability analysis increases the likelihood that $\varphi_{correct}$ and $\varphi_{violated}$ are true.

Definition 3. *(**Patch validation in invariant-based APR**). Let P be a program containing bug b and T be a test suite containing at least one failing test and one passing test. Let also pt be a plausible patch that makes P passes all test cases in T. The validity of patch pt can be formally checked as follows*

$$validity(pt) = V(pt, \varphi_{correct}) \wedge \neg V(pt, \varphi_{violated}) \tag{2}$$

where $V(pt, \varphi_{correct}) \in \{true, false\}$ and that the tool's response depends on whether the specification is fulfilled or violated in the program being examined.

To boost confidence in the validity of the resulting patch, we opt to check patches against both $\varphi_{correct}$ and $\varphi_{violated}$. However, to lower the cost of calling the verifier V against each candidate patch, we aim to implement a three-step patch validation method that uses the test suite first and the program verifier afterwards. Generating plausible patches is done in the first step using test cases. Second step involves formally checking plausible patches against the set of bad patterns (property $\varphi_{violated}$). Patches that pass the first two steps are checked against the set of good patterns (property $\varphi_{correct}$) in the third step.

3 Fixing Performance Bugs Using Invariant-based APR

Performance bugs are programming errors that cause significant performance degradation - lead to low system throughput. Experience has shown that many commercial software that is widely used suffer from performance problems [13, 6, 10]. Therefore, there is a need to develop a rigorous repair framework for performance bugs that ensures efficiency gain without compromising functionality.

One unique characteristic of performance bugs comparing to functional bugs is that performance bugs do not affect the functionality of the program (i.e., the program is *semantically correct but inefficient*) and thus the intended behavior of the program can be automatically deduced using an invariant inference tool.

This section describes an invariant-based APR system for performance bugs and demonstrates how it may be applied to handle performance bugs by producing patches that ensures efficiency improvement without sacrificing functionality.

3.1 Invariant-based Repair Framework for Performance Bugs

In this section we describe an invariant-based repair framework for handling performance bugs. The framework consists mainly of the following components:

1. a set of passing tests (tests that lead to fast runs),
2. a set of failing tests (tests that lead to slow runs),
3. runtime monitor to keep track of the program's execution time and differentiate between fast and slow runs, and
4. an automated invariant inference tool (Daikon or CPAChecker) and automated invariant verification tool (PVS, Z3 solver, or CPAChecker).

We now turn to discuss how we define the notions of passing and failing tests and the process of generating and validating patches for performance bugs.

Passing and failing tests for performance bugs: Performance bugs do not produce debugging information at runtime: they do not produce crashes, exceptions, or incorrect results. We therefore use a runtime monitor with a predefined timer to redefine the concepts of passing and failing tests. We consider test cases that lead to *fast runs* as passing tests while test cases that lead to *slow runs* as failing tests. A repair that transforms slow runs into fast runs while preserving the desired behavior of the original program is considered as a valid repair.

Patch generation strategy for performance bugs: Since we deal with a semantically correct but inefficient program, an efficient version of the program can often be created by restructuring the original program's basic components. Our preliminary analysis demonstrates the effectiveness of genetic repair tools, such as GenProg, in dealing with performance bugs. This suggests that programs with performance bugs can be fixed by relatively simple changes. For instance, various performance bugs can be fixed by using mutation operators like move, swap, delete, and insert employed by genetic repair programs. Consequently, we aim to combine our repair framework with genetic-based patch generation tools.

Patch validation for performance bugs: It should be noted that invariant inference tools can also be used to derive predicates related to the non-functional attributes of the program. This can be achieved by adding extra non-functional variables to the program being repaired. Suppose we have a program P with a set of variables V and that P containing a performance bug. We need to check whether the generated plausible patch for program P fixes the performance bug without introducing new functional bug. To do so, we first generate and validate predicates related to the efficiency attributes of the program, as described below.

1. Add a fresh variable `nfv` whose value has no impact on the behavior of P. The type of performance bug that is being handled determines how `nfv` is used to model the efficiency of the program. However, for the loop programs we consider, `nfv` acts like a counter that is incremented once for each iteration. In other words, the number of loop iterations serves as a model for efficiency.

2. Use the invariant detection tool D to infer the numerical invariants $\mathcal{I}(P,\ \texttt{nfv})$ and $\mathcal{I}(pt,\ \texttt{nfv})$ for the original and plausible patched version, where $\mathcal{I}(P,\ \texttt{nfv})$ represents the collection of invariants in program P involving variable nfv.
3. Compare the numerical predicates in $\mathcal{I}(P,\ \texttt{nfv})$ and $\mathcal{I}(pt,\ \texttt{nfv})$ to determine whether the patched version pt is more efficient than original program P.

For simplicity reasons, we assume we deal with a program with a single loop. The number of loops in the analyzed program, however, determines how many more variables are needed. The invariant inference tool D is thus used to infer invariants on $(V \cup \{\texttt{nfv}\})$. We then distinguish the following types of predicates:

- $\mathcal{I}(P, V)$: predicates related to the program's functionality, and
- $\mathcal{I}(P,\ \texttt{nfv})$: predicates related to the program's efficiency.

Using the generated predicates, one can check the validity of patch pt as follows

$$validity(pt) = \textsc{SemaEq}\ (\mathcal{I}(P,V), \mathcal{I}(pt,V))\ \wedge \textsc{PredSm}\ (\mathcal{I}(pt,\ \texttt{nfv}), \mathcal{I}(P,\ \texttt{nfv})) \qquad (3)$$

where SemaEq is a Boolean operation that checks whether the given sets of invariants are semantically equivalent and PredSm is a Boolean operation that checks whether the upper bound in the predicate related to the patched version is smaller than the upper bound in the one related to the original program.

We now describe two formal procedures to verify the validity of plausible patches (specification (3)) using the available program verification tools.

1. *Daikon-PVS*: In this patch validation procedure, Daikon is used to generate predicates related to the functional and efficiency attributes of programs P and pt. In the event that $\mathcal{I}(P,V)$ and $\mathcal{I}(pt,V)$ (i.e., predicates related to functional attributes) are not identical, it may be necessary to examine both equivalence and implication relations between the predicates in those sets in order to determine whether P and pt are semantically equivalent. By querying the theorem prover PVS, this task can be accomplished.
2. *CPAChecker-PVS*: One interesting feature in CPAChecker is that it produces correctness witnesses in GraphML format and in those witnesses, one can find the invariants of the analyzed program. This feature can be utilized to generate the set of invariants in both the original program and corresponding plausible one. In case that the invariants generated for both programs are not identical, it may be necessary to examine both equivalence and implication relations between the predicates in the two sets by invoking the prover PVS.

3.2 Fixing real-world performance bugs using invariant-based APR

In this section, we show how invariant-based APR can be used to handle real-world performance bugs. For space reasons, we only consider one interesting example of performance bugs (see Listing 1). The bug is based on a real-world flaw that occurred in Apache and has also been analyzed by other researchers [14].
Analysis of the program in Listing 1: The program aims to determine whether a given (target) string is contained within another (source) string. If

```
1  int found = -1;
2  while (found < 0 ) {
3    // Check if string source[] contains target[]
4    char first = target[0];
5    int max = sourceLen - targetLen;
6    for (int i = 0; i <= max; i++) {
7      // Look for first character.
8      if (source[i] != first) {
9        while (++i <= max && source[i] != first);
10     }
11     // Found first character
12     if (i <= max) {
13       int j = i + 1;
14       int end = j + targetLen - 1;
15       for (int k=1; j<end && source[j]==target[k]; j++, k++);
16       if (j == end) {
17         /* Found whole string target. */
18         found = i;
19         break;
20       }
21     }
22   }
23   // append another character; try again
24   source[sourceLen++] = getchar();
25 }
```

Listing 1. A challenging performance bug found in Apache

the target string is found in the source string, the program sets the variable `found` to the index of the target string's first character. But there is a significant performance flaw in the program: when the target string is at the start of the source string, the run is fast, and the program stops almost instantaneously. On the other hand, the run is slower and takes longer to finish when the target string is closer to the end of the source string. This is mostly because there will be a significant increase in the number of redundant computations. The fault is that the initialization statement of the control variable `i` of the for loop at line 6 should be placed outside the scope of the main while loop just after the initialization of the variable `found`. The longest run that we reported occurs when the source string has a length of 10^7 characters, and the target is a single character that is present at the end of the source string. In this instance, the program runs for 30 hours before terminating and producing the correct results.

3.3 Results and analysis

To handle the performance bug at Listing 1, we select two APR tools: the search-based repair tool GenProg [7] and the semantic-based repair tool FAngelix [16].

These are general-purpose repair tools for C code that can be used to fix a range of program bugs, including loop program bugs. While GenProg successfully generated a plausible patch, FAngelix was unable to produce a plausible one. To avoid doing repetitive calculations in the original program, GenProg moved the initialization statement of the variable `i` outside of the for loop at line 6. In other words, the program starts with the initialization statement of the variable `i` in the patched version. In this case, the generated patch passes the test cases since `i` is no longer being set to 0 every time the loop receives a new character.

To check the validity of the plausible patch generated by GenProg, we run the tool Daikon and compare the functional and efficiency predicates obtained for both the original program and the plausible patch. Daikon generates the same set of invariants w.r.t. functional variables (i.e., both the original and the patched versions have the same invariants w.r.t. program variables.) This demonstrates that the patch maintains the functional behavior of the original program.

Listing 1 contains four loops: the while loop at line 2, for loop at line 6, while loop at line 9, and for loop at line 15. To evaluate the efficiency of the original and patched programs, it is sufficient to calculate the upper bound on the number of iterations, as the patch does not modify the logic of any of the loops by adding or removing an operation. That is, each iteration of the four loops in both programs involves the same number of operations. We therefore add four iteration counters ($cnt_2, cnt_6, cnt_9, cnt_{15}$) to model the efficiency of each loop, where the index of the counter corresponds to the line number of the loop being analyzed. For instance, the counter cnt_2 is initially set to zero and advanced by one whenever the loop at line 2 is run. We make the following observations when analyzing the efficiency predicates for both the buggy and patched versions:

- Invariants generated for the counter variables cnt_2 and cnt_{15} in the buggy and patched versions are the same. This indicates that the patch does not affect the number of times the loops at lines 2 and 15 are iterated.
- The counter variable cnt_9 only advances in the buggy version and results in the invariant $cnt_9 \leq 500499$. The fact that the patched version no longer employs the while loop at line 9 is a sign of a major improvement.
- Daikon generated the invariant $cnt_6 \leq 1001$ in the buggy version and invariant $cnt_6 \leq 501$ in the patched version. This shows that the loop at line 6 is iterated 50% less times in the patched version than it is in the original code.

The aforementioned findings, along with the fact that the derived functional predicates of both the original and patched versions are identical, boost our confidence about the validity of the generated patch by the tool GenProg.

4 Related Work

Patch overfitting in APR: Several solutions have been developed to alleviate the overfitting problem in APR, such as symbolic specification inference [8], machine learning-based prioritization of patches [1], fuzzing-based test-suite augmentation [5], and concolic path exploration [12]. These solutions rely on limited

incomplete test cases and do not guarantee the general correctness of the patches. Compared to those approaches that generate test inputs, invariant-based APR automatically generates and refines desired invariants that need to be maintained and violated invariants that need to be repaired when modifying code, which makes the approach more reliable than existing repair approaches.

Modern general-purpose APR tools still rely on symbolic execution or concolic execution [9, 12] to discover counterexamples and generate repairs. However, these repair approaches manually inspect to determine whether the generated patches are correct or identical to developer patches, which could be error-prone. Invariant-based APR makes it possible to apply automated verification techniques to alleviate overfitting problem and formally and systematically check the accuracy of generated patches by comparing them to the developers patches.

Handling performance bugs: Several attempts have been made to detect and repair performance bugs in programs using dynamic, static, and hybrid analysis approaches [13, 6, 10]. [10] carried out an empirical investigation into performance bugs and presented several efficiency rules for identifying them. Using dynamic-static analysis techniques, several fix strategies have been developed in [13] to identify and fix performance problems. However, our method is different from previous studies in that it is a more general and rigorous technique that makes use of program invariant to address loop program performance issues and yield reliable patches. Thanks to program invariants, the original program's efficiency can be systematically compared to the patched version.

5 Conclusion and Future Work

We described a novel general-purpose APR system based on the concept of program invariants. Invariant-based APR holds the promise to handle a wider range of bugs and produce more reliable patches than other APR approaches. This is because invariant-based repair systems depend on stronger correctness criteria rather than test suites. We demonstrate the usefulness of leveraging invariants in APR by developing an invariant repair system for performance defects. The preliminary results showed that invariant-based APR can assist in generating valid patches that ensure efficiency improvement without compromising functionality.

Future work: To complete the line of research initiated here regarding invariant-based APR, we identify the following key directions for future work.

- First and foremost, we aim to conduct a thorough empirical analysis to determine how well invariant-based APR handles functional and non-functional defects in programs. This also entails assessing the invariant inference and invariant verification tools that are currently accessible.
- Accurate invariant generation is required to ensure the validity of patches produced by invariant-based APR. We conjecture that reachability analyses can aid with this complex computational task and we aim to combine invariant-based APR with program verification tools that support both invariant generation and refinement such as CPAChecker and PathFinder.

References

1. J. Bader, A. Scott, M. Pradel, and S. Chandra. Getafix: learning to fix bugs automatically. *Proc. ACM Program. Lang.*, pages 159:1–159:27, 2019.
2. D. Beyer. Automatic verification of C and java programs: SV-COMP 2019. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS*, volume 11429, pages 133–155, 2019.
3. D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *Computer Aided Verification*, pages 184–190, 2011.
4. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.
5. X. Gao, S. Mechtaev, and A. Roychoudhury. Crash-avoiding program repair. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 8–18, 2019.
6. G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 77–88. ACM, 2012.
7. C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, Jan. 2012.
8. S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering (ICSE)*, pages 691–701, 2016.
9. S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *International Conference on Software Engineering (ICSE)*, pages 691–701, May 2016.
10. A. Nistor, T. Jiang, and L. Tan. Discovering, reporting, and fixing performance bugs. In *10th Working Conference on Mining Software Repositories (MSR)*, pages 237–246, 2013.
11. Z. Qi, F. Long, S. Achour, and M. C. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 24–36, 2015.
12. R. Shariffdeen, Y. Noller, L. Grunske, and A. Roychoudhury. Concolic program repair. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2021.
13. L. Song and S. Lu. Performance diagnosis for inefficient loops. In *Proceedings of the 39th International Conference on Software Engineering, ICSE*, pages 370–380, 2017.
14. L. Song and S. Lu. Performance diagnosis for inefficient loops. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 370–380, Buenos Aires, Argentina, 2017.
15. W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Software Engineering*, 10(2):203–232, 2003.
16. J. Yi and E. Ismayilzada. Speeding up constraint-based program repair using a search-based technique. *Information and Software Technology*, page 106865, 2022.