Enhancing Automated Program Repair with Deductive Verification

Xuan-Bach D. Le¹, Quang Loc Le⃣, David Lo*, Claire Le Goues†
¹School of Information Systems, Singapore Management University, Singapore
†Singapore University of Technology and Design
⃣School of Computer Science, Carnegie Mellon University
Email: {dbx.le.2013,davidlo}@smu.edu.sg, quangloc_le@sutd.edu.sg, clegoues@cs.cmu.edu

Abstract—Automated program repair (APR) is a challenging process of detecting bugs, localizing buggy code, generating fix candidates and validating the fixes. Effectiveness of program repair methods relies on the generated fix candidates, and the methods used to traverse the space of generated candidates to search for the best ones. Existing approaches generate fix candidates based on either syntactic searches over source code or semantic analysis of specifications, e.g., test cases. In this paper, we propose to combine both syntactic and semantic fix candidates to enhance the search space of APR, and provide a function to effectively traverse the search space. We present an automated repair method based on structured specifications, deductive verification and genetic programming. Given a function with its specification, we utilize a modular verifier to detect bugs and localize both program statements and sub-formulas in the specification that relate to those bugs. While the former are identified as buggy code, the latter are transformed as semantic fix candidates. We additionally generate syntactic fix candidates via various mutation operators. Best candidates, which receives fewer warnings via a static verification, are selected for evolution though genetic programming until we find one satisfying the specification. Another interesting feature of our proposed approach is that we efficiently ensure the soundness of repaired code through modular (or compositional) verification. We implemented our proposal and tested it on C programs taken from the SIR benchmark that are seeded with bugs, achieving promising results.

Index Terms—Automated Repair, Genetic Programming, Deductive Verification, Sound Repair

I. INTRODUCTION

To automate the maintenance process, researchers have recently shown an increased interest in program repair tools [7], [15], [21], [29]. Automated program repair is a process of detecting bugs, localizing bugs, generating fix candidates and validating the fixes. We can broadly classify these tools into two categories. The first class of program repair tools generates fix candidates syntactically. These are tools (i.e. [6], [14], [24], [27], [29], [13], [19]) that repair the program based on a heuristic (and sometimes stochastic, such as guided via a genetic programming approach) traversal of the space of syntactic modifications. These heuristic-based tools have been shown to be as the most effective and scalable ones [14].

The second class of program repair tools generates fix candidates semantically. These are tools (i.e. [4], [7], [15], [21], [26], [20], [12]) that repair programs based on specifications and semantic analysis. Specifications can be a test suite [21], [26], [20], behavioral specification [4] or component specification [7]. Analyses used in these tools include specification inference [15], [21], [20], bounded verification with SAT [4], abstract interpretation [15], and model checking [7]. Bugs are detected when the semantic analysis cannot prove the correctness of a program against its specification. Based on the specification, those tools can synthesize fix candidates such that a repaired program patched with such a fix candidate complies to the specification semantically.

In this work, we propose to combine syntactic search-based and semantic-based techniques to enhance the expressiveness of fix candidates. We present a framework for automated program repair that integrates deductive verification into a genetic programming technique. We focus on programs whose functional specifications (pre-, post-condition, code contracts or assertions) are available and their correctness can be checked by an automated software verification tool. Given a program with its specification, our system detects bugs, identifies the root cause of the bugs by localizing them to relevant program statements and sub-formulas of the specification, generates semantic and syntactic fix candidates, and evolves candidates that receive fewer warnings from static verification until a candidate complying with the specification is found. Since the validation is based on sound deductive verification, our approach guarantees the soundness of repaired programs with respect to provided specifications. At the same time, we achieve scalability through modular (compositional) verification.

More concretely, our approach is a combination of three techniques:

- **Structured specification.** We use a structured specification mechanism [3] that is capable of capturing expressive program requirements. This structured specification allows verification to be done efficiently.

- **Deductive verification.** We use a Hoare-style verification system, which abstracts program code to logic formulas following symbolic execution paradigm. Function calls are compositionally analyzed through their pre- and post- specifications. The correctness of a program is reduced to the correctness of generated verification conditions (i.e. logical implications). Localization is performed at the proof level [9] with a mapping between transformed abstraction (formulas) and program source code.
• Genetic programming. Since automated program repair using genetic programming uses the test cases to compute the fitness of a candidate solution, we redesign the algorithm to work with deductive verification and logic specifications. Leveraging genetic programming, our repair technique is general and able to deal with not only incorrect but also missing source code implementation.

The integration of genetic programming and deductive verification to automated program repair is the primary novelty of our approach. To the best of our knowledge, we are the first one to propose such an integration for automated program repair. Additionally, we propose path-sensitive bug localization by pairing states at each program location with control-flow variables, i.e., the variables of test conditions of if and loop statements, to enable our system to generate semantic fix candidates for buggy if and loop conditions.

We have implemented a prototype of our proposed approach on top of GenProg [29] (as the genetic programming component) and HIP/SLEEK [1, 9] (as the deductive verification component). We have performed a preliminary evaluation of our prototype on a set of ten C programs from the SIR benchmark that are seeded with bugs. Our experiment results show that our approach can automatically produce sound repairs for incorrect and missing implementation errors (including security defects, i.e., buffer overflow) in less than seven minutes for all bugs, and outperforms recent state-of-the-art semantics-based repair tool named Angelix [20], in terms of the number of bugs correctly fixed.

The remainder of the paper is structured as follows. We describe a motivating example to illustrate our proposed approach in Section II followed by background in Section II. Section IV describes the details of our proposed approach. Section V presents the results of a preliminary evaluation of our approach. Section VI discusses related work, followed by conclusion and future work in Section VII.

II. A MOTIVATING EXAMPLE

In Listing 1, we show a code snippet from a function addstr in the replace program taken from the SIR benchmark [2].

In the code snippet, we use requires and ensures keywords to express the pre-condition and post-condition of the function. In the requires and ensures clauses, the semantic of int *j, is specified as j→int_ref(j_val) where j is a pointer pointing to an object with value j_val. Primed variables denote the value of function parameters at the return point of the function. For example, j_val' denotes the value of j_val after the execution of function addstr.

The requirement of the else branch of addstr is specified in line 5. The specification in line 5 indicates two important constraints. First, the value of j after exiting the function addstr, denoted by j_val', is increased by one. Second, the value of outset[j]' after exiting the function addstr, denoted by outset'[j_val'−1]', is equal to c. However, the else branch has been incorrectly implemented. As the value of j is increased (line 14) before the array update (line 15), the array outset may be overflowed. For this example, our verification component detects a bug, and localizes the root cause of the bug as statements in lines 11, 14 and 15 since they violate the constraint outset'[j_val'−1]'=c (at line 5) of the specification. Based on the violated specification, our system generates a semantic fix candidate which replaces outset+[j]=c with outset+[j−1]=c. Our genetic programming component proposes another syntactic fix candidate for this bug by just swapping the statements at line 14 and line 15. These two fix candidates are then soundly validated by the verifier. This example demonstrates that combining both syntactic and semantic candidates condenses the correct repairs within a huge space of possible candidates.

III. BACKGROUND

A. Program Repair with GenProg

GenProg uses genetic programming to guide the search for a valid repair of a buggy program [28]. GenProg takes as input a buggy program along with a set of test cases. The repair process goes through two main phases: bug localization and valid patch finding. At the first phase, suspicious areas containing statements that are likely buggy are identified by running the program against the test cases. These suspicious areas are then the targets for the second phase which will generate fix candidates.

In the second phase, GenProg leverages genetic programming to generate fix candidates and search for a valid repair among them. To create fix candidates (aka variants), GenProg primarily uses mutation operators to syntactically modify the original program in the suspicious areas. There are four mutation operators: add, replace, delete and swap. These mutations are applied to suspicious statements of the original program to generate various variants that are possible candidate repairs. The suitability of generated candidates is then computed by running the candidates against test cases, i.e., candidates passing more test cases have a higher suitability score. Better candidates with higher suitability scores, are selected and carried over to form a new generation of variants. This process is repeated many times until a valid repair which
passes all the test cases is found or when a limit is reached (i.e., a time limit, or a maximum number of generations).

B. Program Verification with HIP/SLEEK

Specification Language. We use structured specification (from [3]) which supports case analysis of the form below:

\[
Y ::= \text{requires } \Phi \quad | \quad \text{case } \{ \pi_1 \Rightarrow Y_1; \ldots; \pi_n \Rightarrow Y_n \} \quad | \quad \text{ensures } \Phi
\]

whereby \( \Phi \) is a separation logic formula [22]. A separation logic formula is a conjunction of a heap formula and a pure (non-heap) formula. A heap formula is a spatial conjunction (+) of empty heap assertions (emp), points-to assertions (\( \rightarrow \)), and user-defined predicates [1]. A pure formula is a first-order logic formula capturing an expressive constraints of numerical and bag/set domains. In HIP/SLEEK system, structured specification with separation logic can be used to efficiently and effectively verify functional correctness [3]. Furthermore, such a specification can be automatically inferred via second-order bi-abduction [3].

In the case block of the specification, \( \pi_1, \ldots, \pi_n \) are guards corresponding to test conditions of if statements. Guards are pure formulas without quantifiers. For a sound and complete verification, these guards need to satisfy two conditions: disjointness (i.e. \( \pi_i \land \pi_j = \text{false} \quad \forall \ i, j \in \{1, \ldots, n\} \) and \( i \neq j \)), and universe (i.e. \( \pi_1 \lor \pi_2 \lor \ldots \lor \pi_n = \text{true} \)). We exploit these requirements to efficiently perform verification when we validate fix candidates.

Modular Verification. HIP/SLEEK is a modular verification system. It verifies a set of dependent functions bottom-up and in isolation from other dependent functions. To verify a function, its source code is symbolically executed and abstracted into separation logic formulae. Functional correctness is reduced to the correctness of verification conditions which are generated during the symbolic execution. For example, verification conditions generated at function exits for postconditions proving have the following form:

\[
\text{precondition } \land \text{code } \land \text{abstract } \vdash \text{postcondition}
\]

These verification conditions are discharged by SLEEK, a separation logic entailment procedure. In the case of a bug being found, i.e., a verification condition cannot be proven, HIP is able to identify the root cause of the bug by localizing sub-formulas in the specification and program statements that are relevant to the bug [9].

IV. PROPOSED APPROACH

The primary goal of our approach is to integrate deductive verification into genetic programming to generate both semantic and syntactic fix candidates as well as to produce sound repairs in a modular fashion. Fig. 1 depicts the workflow of our approach. There are three main phases: bug detection/localization, fix candidate generation and fix candidate validation/selection. These three phases are described in the following subsections.

A. Detecting and Localizing Bugs

The goals of this phase are to identify potential buggy statements and uncover parts of the specification that are violated.

This phase takes as input a program and its specification. The HIP/SLEEK deductive program verifier checks the program against the specification. If an error is detected, the verifier identifies the root cause of the error which includes suspicious statements and the corresponding violated parts of the specification. For generating more expressive fix candidates, we enhance the HIP/SLEEK error calculus presented in [9] with path-sensitive localization. We implement the new feature by adding variables \( v \) in test conditions of each if statement to every downstream statements of the corresponding then and else branches. We also add variables \( v \) in test conditions of each loop statement to every downstream statements to body of the loop. When a downstream statement is a potential root cause of the error, so is \( v \). This allows us to potentially identify buggy if- and loop-conditions.

The potential buggy statements and parts of the specification that are violated are passed to the next phase for generating fix candidates.

B. Generating Fix Candidates

In this phase, we first generate semantic fix candidates from the violated sub-formulas of the specification. We generate two kinds of semantic fix candidates. If the violated sub-formulas belong to either preconditions or guards or case blocks, we generate fix candidates corresponding to expressions of if statements. Otherwise, we generate fix candidates corresponding to assignment statements. We additionally generate syntactic candidates by employing various mutations operators borrowed from GenProg, e.g., append, delete and swap. These fix candidates are then evolved through genetic programming to create populations of variants that may contain a valid repair.
The reason for incorporating syntactic and semantic candidates is that these candidates could help one another in tandem to condense the search space into more valuable candidates overall. For example, the search space with syntactic repair candidates alone is sometimes very sparse [10], and real fixes could lie beyond the syntactic candidates. Semantic candidates, however, could cultivate the search space with more useful candidates. The intuition is that a specification already captures the primary functionality of a program, and thus the concrete code extracted from the specification would help in patching the current buggy implementation of the program.

C. Validating and Selecting Fix Candidates

In this phase, we validate the variants generated by the previous phase. We use the HIP/SLEEK system to verify each of the variants in a modular fashion. We exploit the modularity of deductive verification to compositionally verify the variants. Particularly, we only verify the functions that contain suspicious statements discovered by the bug localization phase.

After each candidate has been verified, a fitness score, which defines how good a repair candidate is, is computed. We define the fitness score as the number of warnings produced by the verifier. Better candidates, whose fitness scores are lower, are selected and carried over to form a new population of mutants (aka. a new generation). This process is repeated many times until a valid repair which verifies the fitness score is found or when some limits are reached (i.e., the time limit is reached, or the maximum number of populations is reached).

V. PRELIMINARY EVALUATION

Experimental Setup. We experiment on programs from the SIR benchmark [2], which has been extensively used in research in software engineering, particularly by studies that propose new testing and fault localization approaches, e.g., [5], [18]. We reused SIR program specifications constructed by Le et al. [9] and manually injected seeded bugs to the original programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Mutated Location</th>
<th>LOC</th>
<th>Time</th>
<th>Bug Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniq</td>
<td>gline_loop</td>
<td>74</td>
<td>0.5</td>
<td>Incorrect</td>
</tr>
<tr>
<td>replace</td>
<td>addstr</td>
<td>855</td>
<td>2.8</td>
<td>Missing</td>
</tr>
<tr>
<td>replace</td>
<td>stclose</td>
<td>855</td>
<td>2.15</td>
<td>Missing</td>
</tr>
<tr>
<td>replace</td>
<td>stclose</td>
<td>855</td>
<td>2.2</td>
<td>Incorrect</td>
</tr>
<tr>
<td>replace</td>
<td>locate</td>
<td>855</td>
<td>2.5</td>
<td>Incorrect</td>
</tr>
<tr>
<td>replace</td>
<td>patsize</td>
<td>855</td>
<td>0.5</td>
<td>Incorrect</td>
</tr>
<tr>
<td>replace</td>
<td>esc</td>
<td>855</td>
<td>2.14</td>
<td>Incorrect</td>
</tr>
<tr>
<td>scheduled3</td>
<td>dupp</td>
<td>693</td>
<td>0.43</td>
<td>Incorrect</td>
</tr>
<tr>
<td>print_tokens</td>
<td>ncl</td>
<td>1002</td>
<td>6.25</td>
<td>Missing</td>
</tr>
<tr>
<td>tcas2</td>
<td>IBC</td>
<td>302</td>
<td>0.15</td>
<td>Incorrect</td>
</tr>
</tbody>
</table>

Threats to Validity. Threats to external validity relate to errors in our implementation and experiment. We have put an effort to ensure that our implementation and experiment are correct, however, there could be errors that we did not notice. Threats to external validity relate to the generalizability of our findings. In our preliminary experiment, we have only evaluated our approach on 10 buggy programs from SIR benchmark with specifications constructed by Le et al [9] (a co-author of this paper). In the future, we plan to reduce this threat to external validity by evaluating our approach with more programs and different kinds of bugs. Also, our approach requires complete specifications in order to completely ensure the soundness of generated patches. We expect that employing advances in specification inference and mining could help reduce this burden [10]. Future work would be to use the history of bug fixes to mine specifications of past patches and inform the future patch generation process in a way that generated patches are likely to conform to desired behaviours.
Syntactic-based approaches. Le Goues et al. proposed GenProg, the first search-based automated patch generator [29]. [14]. To improve GenProg, Weimer et al. proposed a cost model and used program equivalence to reduce fix search space [27]. Qi et al. replaced genetic programming with random search to reduce the number of test case executions [24]. Kim et al. manually constructed fix templates learned from many human-written patches [6], and applied these templates to fix bugs. Long et al. presented Prophet, a tool that automatically learn from human-written patches to fix new bugs [17]. Le et al. introduced HDRP that uses bug fix history to assess suitability of patches [13].

While the above approaches use dynamic analysis (i.e., testing) for detecting/localizing bugs and validating repaired programs, we use static analysis (i.e., deductive verification). Furthermore, our approach integrates syntactic and semantic-based approaches, by combining semantic fix candidates constructed by analyzing a program and its specification with syntactic fix candidates generated by genetic programming.


Like these approaches, we make use of specification and static analysis technique for automated program repair. Different from them, we employ Hoare-style verification combined with genetic programming to integrate syntactic and semantic-based approaches.

VII. CONCLUSION AND FUTURE WORK

In this work, we propose a novel program repair technique, which marries the strengths of static verification and genetic programming, to produce sound repairs. Our approach uses the structured specification of a program together with genetic programming to evolve a buggy program until the bugs are gone. To enhance the efficiency of genetic programming, we generate high-quality fix candidates which are inferred using semantic analysis of the program and its specification. We employ modular deductive verification to generate fix candidates and validate the fixes. We have implemented a prototype and evaluated it on ten C programs from SIR benchmark with promising results.

We plan to extend the empirical evaluation of our approach with additional programs and more bugs of various types. We also plan to handle programs manipulating complex data structures, e.g., singly-linked list, binary tree, AVL tree, etc. We could also employ machine learning techniques to help traverse the search space of possible candidates, e.g., predict correct patches among the search space like prediction tasks proposed in [17]. [25]. We also plan to systematically compare our proposed technique with more APR approaches.

Acknowledgements. Quang Loc is partially supported by grant T2MOE1303. This research was funded in part by the US Air Force (#FA8750-15-2-0075) and by the US Department of Defense through the Systems Engineering Research Center (SERC) (H98230-08-D-0171).

REFERENCES