Debugging Declarative Models in Alloy

Guolong Zheng
University of Nebraska-Lincoln
Lincoln NE, USA
Email: gzheng2@unl.edu

Hamid Bagheri
University of Nebraska-Lincoln
Lincoln NE, USA
Email: bagheri@unl.edu

ThanhVu Nguyen
University of Nebraska-Lincoln
Lincoln NE, USA
Email: tnguyen@cse.unl.edu

***Abstract*—Debugging, which involves both fault localization and bug repair, is critical for developers to identify and remove bugs in a program. Most existing debugging research techniques focus on imperative programs (e.g., C and Java) and rely on test suite to analyze correct and incorrect executions of the program to identify and repair suspicious statements.**

We propose a new debugging framework for models written in a declarative language, where the models are not "executed", but rather converted into a logical formula solvable using a constraint solver. In recent work, we developed a fault localization tool that takes as input an Alloy model consisting of a violated assertion and returns a ranked list of suspicious expressions contributing to the violation. Preliminary results show that the fault localization tool is accurate, useful, and scales to complex, real-world Alloy models.

In this work, we propose a new repair technique and tool that can be integrated with our fault localization tool or used as a stand-alone tool. We aim to automatic repair bugs violating given assertions in Alloy models. We plan to adopt guided search and pattern-based repair techniques from imperative automatic program repair and modify DFA learning algorithms to synthesis repairs.

I. Introduction

Alloy [1] is a declarative language for modelling the structure and behaviour of complex hardware and software systems. It has been used in a wide range of applications, such as program verification [2], test case generation [3], [4], network and security [5], [6], IoT [7] and Android security [8], [9], and design tradeoff analysis [10]. For instance, Taco [2] verifies a Java program by translating it to Alloy and converting pre/post conditions to assertions.

Given a model written in the Alloy language, a developer can write an assertion or a predicate to check if the model violates or satisfies some property. Alloy then translates the model and the assertion or predicate to one SAT formula and uses an off-the-shelf SAT solver to find a solution that violates the assertion or satisfies the predicate.

Similar to writing programs in an imperative language, such as C or Java, users can make mistakes when writing Alloy models, especially those that capture complex systems with non-trivial behaviours. Moreover, the user might not be familiar with the syntax and semantics of the Alloy language and can make subtle mistakes, hard to detect by a novice developer. Hence, debugging techniques and tools, such as fault localization techniques to identify buggy code in programs, and automated repair tools to fix the error can help the users write correct Alloy code and thus help increase the uses and adoption of Alloy and formal methods in general.

Compared to imperative languages, there are fewer debugging techniques developed for declarative languages. Indeed, AlloyFL [11] and ARepair [12] are perhaps the only fault localization and repair tool available for Alloy as of today. The key idea of AlloyFL is to use "unit tests," where a test is a predicate that describes an Alloy instance to encode expected behaviours and compute suspicious expressions in an Alloy model that fails these tests. To compute these suspicious expressions, AlloyFL uses mutation testing [13], [14] and statistical debugging techniques [15], [16], i.e., it mutates expressions, collects statistics on how each mutation affects the tests, then uses this information to assign suspicious scores to expressions. ARepair builds on top of AlloyFL and repairs bugs based on generate-and-test technique, where it synthesizes new patches based on AlloyFL results and check if the patch passes all test cases.

While AlloyFL and ARepair are the pioneer in Alloy debugging and are promising, they rely on the assumption of the availability of AUnit tests [17], which is not common in the Alloy setting. Indeed, instead of writing test cases, Alloy users write assertions to describe the desired property and let the Alloy Analyzer search for specific instances that violate the property. Moreover, it is unclear how many test cases are needed or how good they must be for AlloyFL and also ARepair to be effective (e.g., in the AlloyFL evaluation [11] the number of tests range from 30 to 120). Another concern is the runtime for ARepair is relatively large even for small Alloy models due to heavy Alloy calls, and several kinds of bugs can't be fixed due to large search space or missing synthesis guidance. For example, ARepair fails to repair bugs which require adding new facts.

To address this state of affairs and to improve the quality of Alloy development, we propose a new debugging approach that automatically finds and fixes errors in Alloy models violating assertions instead of "unit tests".

In recent work [1], we developed a new technique and tool that finds faulty expressions causing an assertion violation in an Alloy model. It utilizes Alloy to generate pairs of a counterexample and satisfying instance and uses their differences to identify suspicious expressions. We evaluated the technique on a benchmark consisting of a suite of buggy models from ARepair [12] and two large real-world models. Experimental results on real-world Alloy models corroborate that the tool is

[1]This work is being submitted for review.

able to consistently rank buggy expressions in the top 1.7% of the suspicious list.

In this paper, we propose to extend our fault localization technique and a new algorithm to automatically repair bugs violating assertions in Alloy. We plan to explore the idea of template-based repair [18], in which commonly used repair patterns are summarized or learned from previous repair patches and are applied to repair new bugs. Template-based repair is a well-studied area in automated program repair [18]–[20] and has shown effective repairing performance. However, it hasn't been explored in declarative languages like Alloy due to language difference and lack of repair patterns. Based on our preliminary results (e.g., able to repair bugs that couldn't be repaired by random search), we believe adopting template-based repair to Alloy would lead to a promising repair performance.

II. Fault Localization

In recent work (in submission), we developed a fault localization tool for Alloy. It takes as input an Alloy model consisting of some violated assertion and returns a ranked list of suspicious expressions contributing to the assertion violation.

a) Fault Localization: The key idea is to analyze the differences between *counterexamples* (instances of the model that do not satisfy the assertion) and *satisfying instances* (instances that do satisfy the assertion) to find suspicious expressions in the input model.

To achieve this, the tool uses the *Alloy analyzer* to find counterexamples showing the violation of the assertion. Then, it uses a PMAX-SAT solver to find satisfying instances that are *as close as possible* to the counterexamples. Next, it analyzes the differences between the counterexamples and satisfying instances to find expressions in the model likely causing the errors. Finally, it computes and returns a ranked list of suspicious expressions.

We use the Alloy model in Figure 1 to show how the tool works. This model specifies a simple type of binary search tree (BST), adapted from ARepair benchmarks [12], with the following constraints, expressed as fact paragraphs in Alloy. All nodes are reachable from the root (fact `Reachable`); each node is acyclic, has no more than one parent and its left and right child are exclusive (fact `Acyclic`). Finally, the predicate `Sorted` states the sorted property of BST.

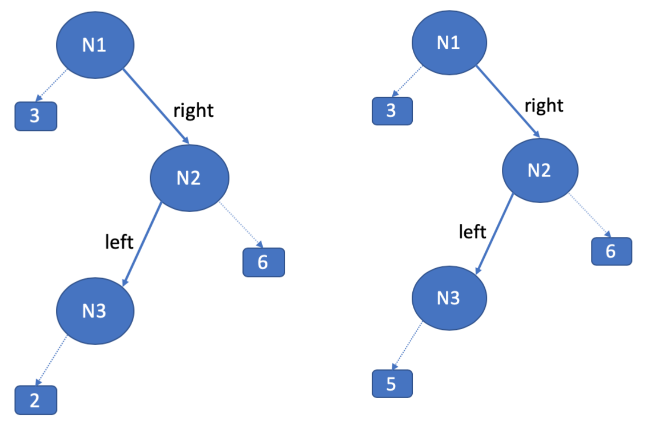
The tool first checks the assertion sorted_tree in the model using the Alloy Analyzer, which returns the counterexample in Figure 2a, where the element of `N1` is larger than the node `N3` in its right subtree. Next, it uses a PMAXSAT solver to generate a satisfying (sat) instance, shown in Figure 2b, that is as minimal and similar to the counterexample as possible. Then it identifies the differences between counterexample and sat, the element of `N3` changing from 2 to 5, and utilizes this information to locates suspicious expressions. For this example, it finds four suspicious expressions with the one line 25 ranked first.

```
1   one sig BinaryTree { root: lone Node }
2   sig Node {
3     left, right: lone Node,
4     elem: Int
5   }
6   fact Reachable {
7     Node = BinaryTree.root.*(left + right)
8   }
9   fact Acyclic {
10    all n : Node {
11      // There are no directed cycles
12      n !in n.^(left + right)
13      // A node cannot have more than one parent.
14      lone n.~(left + right)
15      // A node cannot have another node as both
            its left child and right child.
16      no n.(left) & n.(right)
17    }
18  }
19  pred Sorted() {
20    all n: Node {
21      // All elements in the n's left subtree are
             smaller than the n's elem.
22      all nl: n.left.*(left + right) | nl.elem <
            n.elem
23      // All elements in the n's right subtree
            are bigger than the n's elem.
24      // Fix: "all nr: n.right.*(left + right) |
            nr.elem > n.elem".
25      some n.right => n.right.elem>n.elem
26    }
27  }
28  assert sorted_tree{
29    Sorted => { all sub, root : Node |
30      sub in root.right.*(left + right) implies
            sub.elem > root.elem}
31    }}
32  check sorted_tree
```

Fig. 1: Buggy Binary Search Tree model

b) Results: We implemented the tool, the fault localization technique, in Java and Alloy 4.2, and evaluated its performance on 56 bugs collected from 38 Alloy models collected from ARepair [12] and 8 real-world Alloy specifications modelling surgical robot and Java program verifications. The experimental results show that the fault localization technique is efficient (can handle complex, real-world Alloy models with thousand lines of code within 5 seconds), and accurate (can consistently rank buggy expressions in the top 1.7% of the suspicious list), and useful (can often narrow down the error to the exact location within the suspicious expressions).

c) Integration with Repair: Our previous work on Alloy fault localization shows promising results. However, we predict two main challenges to integrate with our proposed repair technique. One challenge is to reduce false-positive fault localization results. The empirical results in TBar [18] shows that fault localization noise has a significant impact on the performance due to fix patterns are sensitive to the false-positive locations recommended as buggy positions. For 13 out of 56 models, it produces false-positive results, ranking unrelated expressions on top 3. We plan to reduce the false-positive rate by further analyzing the difference to capture the root cause.



(a) Counterexample (b) (PMAX) Sat instance

Fig. 2: Instance Pair.

Another challenge is that besides the faulty location, we also need to capture features of a bug to select the proper repair template. For example, to repair some under constraint bugs, we need to add a completely new constraint, which requires the fault localization tool identifies that no existing expression contributes to the error. We plan to summarize popular Alloy bugs and categorize them into different groups by features selected from the difference between counterexample and satisfying instance and the shape of the buggy expression.

III. Proposed Work: Template-Based Repair

A. Related Work

Automatic program repair is a well-studied area. Many repair techniques follow the generate-and-validate procedure. They first generate a fix candidate by searching or synthesising and then validate the candidate by a set of test cases. Some work [21]–[24] randomly search for a fix. For example, GenProg [22] uses genetic algorithm to repair bugs, and [23] applies random search. Learn-based techniques [20], [25]–[27] learn repair from existing commits or bug reports. For example, Prophet [25] learns repair from human patches, [26] mines bug fix patterns from the projects commits and DeepFix [27] trains a neural network to predict erroneous program locations along with the required correct statements. Synthesis-based technique [28]–[31] captures program semantics and utilizes program synthesis to generate fix. For example, Angelix [32] optimizes symbolic execution to extract constraints to synthesis a fix and SPR [19] synthesis a repair by combining staged program repair and condition synthesis.

Among these techniques, repair patterns are widely used [18]–[20], [33]. PAR [20] manually summarizes concrete fix patterns from human-written patches. Based on static analysis violations, Avatar [33] uses fix patterns to repair bugs. TBar [18] thoroughly investigates popular pattern-based repair and shows they are effective for program repair.

Automatic repair for Alloy, on the other hand, has not been well studied. The only existing work is ARepair [12]. ARepair generates patch by creating holes and synthesizing expressions to fill the holes, then validates the repair by test suites. One main limitation is that the holes are created from and restricted to the original faulty expression, thus fail to fix bugs that require new expressions. To repair more complicated bugs, we propose a new repair technique based on pattern-based repair.

B. Illustrative Example

We use the Alloy model in Figure 1 to illustrate how our repair technique works. We use the assertion `sorted_tree` to check if the tree is a binary search tree. However, the Alloy analyzer disapproves this assertion by providing a counterexample, as shown in Figure 2a, in which the elem of node (`N3`) in right subtree is smaller than the root node (`N1`).

This indicates a bug in the model. The expression on line 25 is underconstrained: it only specifies the elem of the right child is smaller but not all nodes in the right subtree.

To automatically repair this bug, we first locate the bug to the expression in line 25: `all n: Node | some n.right=>n.right.elem>n.elem`. Then we select a template to fix this problem. The templates are selected by the type of difference and by the format of buggy expression(we assume the overall structure of the buggy expression is similar to the correct one). For this example, we choose the following template:

```
all a: X | all b: a.Y | a Op b
```

where X and Y represent expressions and Op represents operators. Our goal is then to fill in with correct expressions and operators.

By comparing the counterexample in 2a and sat in 2b, we find the difference is the `elem` of `N3` changes from 2 in counterexample to 5 in sat. By further analyzing counterexample and sat, we infer that `N3.elem` becomes larger than `N1.elem` while other relations remain the same between counterexample and sat. Then we can describe the difference as `N1.elem < N3.elem`.

We then want to abstract this concrete expression to more general Alloy expression. We first reduce the number of concrete values by replacing `N3` with `N1.right.left` replace N1. Then we abstract `N1` to its type `Node` and get `Node.elem < Node.right.left.elem`.

We then ask Alloy to generate more pairs of counterexamples and sat instances. For example, we would get the following difference abstracts:

- `Node.elem < Node.right.left.right.elem`
- `Node.elem < Node.right.left.left.elem`
- `Node.elem < Node.right.right.left.elem`
- `Node.elem < Node.right.right.left.left.elem`

We then adopt the idea of infering regular expression from string (DFA learning) to infer Alloy expressions. For this example, we infer `Node.right.*(left+right)`. Thus, we fill X with Node and Y with `a.right.*(left + right)` and Op with <, and generate a patch by replacing expression in line 25 with `all a:Node | all b:a.right.*(left+right) | a.elem < b.elem`. We fix the bug by generating no counterexample when checking the assertion.

C. Proposed Repair Algorithm

We propose to adopt the idea of template-based automatic program repair [18], where the repair tool select patches from repair patterns. These patterns are either pre-defined or learned from previous repair patches. For example, one common bug is null pointer deference, and a corresponding repair pattern is checking the pointer is not null before dereferencing the pointer.

We predict three main challenges: (1) The patterns used in imperative language are not suitable for Alloy. For example, there is no "null pointer deference bug" in Alloy. We need new bug taxonomy and repair patterns for Alloy. (2) The complexity of applying patterns. Some repair patterns may lead to overfitting such that the Alloy model is too over-constrained to generate any counterexample. (3) Instantiating repair patterns become challenging due to the complexity of relations between Alloy variable. Unlike imperative language, the variables in a program are declared by certain types, the variables in Alloy are bind to some Alloy expressions which increases the difficulty of instantiating the pattern.

To address these challenges, we will explore the following solutions.

Generate patterns: We plan to summarize some general repair patterns from benchmarks. The challenge is how to make these patterns general enough to cover as many bugs. To leverage this problem, we only define a general sketch as a pattern and try to learn the rest part from buggy expression.

Select patterns based on bug types: The difference between the counterexample and sat is insufficient to select a proper repair pattern, e.g. whether the bug can be fixed by mutating existing expression or adding new expression. To solve this problem, we will collect syntactic and semantic information from the context of the buggy expression, then select proper patterns by this information and the difference.

Apply repair patterns: The main difficulty of applying a template is instantiation, in which we need to infer the bind expression for each variable. We plan to adopt the idea of inferring regular expression for strings, DFA learning algorithm, to infer bind expressions for variables. Another concern is the search space becomes large if there are too many unknowns in templates. We plan to apply partial repair gradually instead of search for a full repair at once. To evaluate the quality of a partial fix, e.g. whether the partial repair is moving towards a correct answer, we plan to use a model counting solver to calculate the number of counterexamples of an assertion and use this number as a guide for the repair, the smaller the better.

Evaluate patch: We already adopt a benchmark from ARepair [12] that contains 38 Alloy models. This benchmark consists of examples that comes with Alloy and homework collected from student assignments. All models in this benchmark are relatively small, but the bugs are diverse and challenging to fix. To further evaluate our work, we plan to collect more examples from online open projects and real-world Alloy models.

D. Preliminary Results for Repair

We are currently exploring new automatic repair ideas and are implementing the pattern-based repair technique. We have summarized several repair patterns and implemented a prototype. We tested the prototype on some small but interesting examples. For example, for a room access Alloy model, where the error is an under constrained bug and ARepair fails to repair this model, as a completely new fact `no Employee.owns` is needed to repair the bug. Our prototype is able to generate the correct patch by determining that the relation `owns` never starts with `Employee` and using the template `no X.Y`.

IV. Expected Contributions

In summary, our proposed contributions are:

Summary of Common Alloy Bugs and Repair Patterns: We will summarize common Alloy bugs and their corresponding repair patterns. We believe this is the first work categorizing Alloy bugs. This bug taxonomy can be further explored in research on Alloy debugging.

Automatic Repair Technique for Alloy: We will develop a novel automatic repair technique for Alloy. The repair technique tries to repair Alloy bugs violating assertions and doesn't need test suites.

Collection of Alloy Benchmarks: We plan to collect practical Alloy models from online open source repositories. In our previous work, we have collected two real-world Alloy models: surgery robot models written by humans and Java verification models generated by the machine. Our benchmark will contain diverse Alloy models and can be used by other researchers for repairing Alloy models.

V. Conclusion

Alloy is a powerful tool and has been used in a wide range of applications, such as program verification, test case generation, network and security, IoT and Android security, and design tradeoff analysis. To better assist Alloy developers, we are working on a debugging technique automatically identifying and repairing bugs for Alloy models with assertions.

In recent work, we developed a new fault localization approach for declarative models written in Alloy. The approach is based on the insight that expressions in an Alloy model that likely cause an assertion violation can be obtained by analyzing the counterexamples, unsat cores, and satisfying instances from the Alloy Analyzer.

In this paper, we propose a new program repair technique for Alloy that relies on assertions to fix bugs. We plan to adopt template-based repair to Alloy. The preliminary results show that this method is able to fix new bugs that previous work couldn't fix.

References

[1] D. Jackson, "Alloy: A lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, p. 256–290, Apr. 2002. [Online]. Available: https://doi.org/10.1145/505145.505149

[2] J. P. Galeotti, N. Rosner, C. G. López Pombo, and M. F. Frias, "Analysis of invariants for efficient bounded verification," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 25–36. [Online]. Available: https://doi.org/10.1145/1831708.1831712

[3] P. Abad, N. Aguirre, V. S. Bengolea, D. Ciolek, M. F. Frias, J. P. Galeotti, T. Maibaum, M. M. Moscato, N. Rosner, and I. Vissani, "Improving test generation under rich contracts by tight bounds and incremental SAT solving," in *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*. IEEE Computer Society, 2013, pp. 21–30. [Online]. Available: https://doi.org/10.1109/ICST.2013.46

[4] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in gui testing of android applications," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 559–570.

[5] F. A. Maldonado-Lopez, J. Chavarriaga, and Y. Donoso, "Detecting network policy conflicts using alloy," in *Proceedings of the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z - Volume 8477*, ser. ABZ 2014. Berlin, Heidelberg: Springer-Verlag, 2014, p. 314–317.

[6] N. Ruchansky and D. Proserpio, "A (not) nice way to verify the openflow switch specification: Formal modelling of the openflow switch using alloy," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, p. 527–528, Aug. 2013. [Online]. Available: https://doi.org/10.1145/2534169.2491711

[7] M. Alhanahnah, C. Stevens, and H. Bagheri, "Scalable analysis of interaction threats in iot systems," ser. ISSTA '20, 2020.

[8] H. Bagheri, J. Wang, J. Aerts, and S. Malek, "Efficient, evolutionary security analysis of interacting android apps," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 357–368.

[9] H. Bagheri, E. Kang, S. Malek, and D. Jackson, "A formal approach for detection of security flaws in the android permission system," *Formal Aspects of Computing*, vol. 30, no. 5, pp. 525–544, 2018. [Online]. Available: https://doi.org/10.1007/s00165-017-0445-z

[10] H. Bagheri, C. Tang, and K. Sullivan, "Trademaker: Automated dynamic analysis of synthesized tradespaces," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 106–116.

[11] K. Wang, A. Sullivan, D. Marinov, and S. Khurshid, "Fault localization for declarative models in alloy," *CoRR*, vol. abs/1807.08707, 2018. [Online]. Available: http://arxiv.org/abs/1807.08707

[12] K. Wang, A. Sullivan, and S. Khurshid, "Arepair: a repair framework for alloy," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 103–106.

[13] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," 03 2014, pp. 153–162.

[14] M. Papadakis and Y. Le Traon, "Metallaxis-fl: Mutation-based fault localization," *Softw. Test. Verif. Reliab.*, vol. 25, no. 5?7, p. 605?628, Aug. 2015. [Online]. Available: https://doi.org/10.1002/stvr.1509

[15] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, ser. TAICPART-MUTATION '07. USA: IEEE Computer Society, 2007, p. 89–98.

[16] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, Aug. 2011.

[17] A. Sullivan, K. Wang, and S. Khurshid, "Aunit: A test automation tool for alloy," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 398–403.

[18] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 31–42. [Online]. Available: https://doi.org/10.1145/3293882.3330577

[19] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 166–178. [Online]. Available: https://doi.org/10.1145/2786805.2786811

[20] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 802–811.

[21] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 364–374.

[22] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[23] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 254–265. [Online]. Available: https://doi.org/10.1145/2568225.2568254

[24] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *2010 Third International Conference on Software Testing, Verification and Validation*, 2010, pp. 65–74.

[25] F. Long and M. Rinard, "Automatic patch generation by learning correct code," *SIGPLAN Not.*, vol. 51, no. 1, p. 298–312, Jan. 2016. [Online]. Available: https://doi.org/10.1145/2914770.2837617

[26] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 213–224.

[27] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," 2017. [Online]. Available: https://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603

[28] P. Liu and C. Zhang, "Axis: Automatically fixing atomicity violations through solving control constraints," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 299–309.

[29] P. Liu, O. Tripp, and C. Zhang, "Grail: Context-aware fixing of concurrency bugs," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 318–329. [Online]. Available: https://doi.org/10.1145/2635868.2635881

[30] Y. Lin and S. S. Kulkarni, "Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 237–247. [Online]. Available: https://doi.org/10.1145/2610384.2610398

[31] D. Gopinath, M. Z. Malik, and S. Khurshid, "Specification-based program repair using sat," in *Tools and Algorithms for the Construction and Analysis of Systems*, P. A. Abdulla and K. R. M. Leino, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 173–188.

[32] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 691–701. [Online]. Available: https://doi.org/10.1145/2884781.2884807

[33] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandè, "Avatar: Fixing semantic bugs with fix patterns of static analysis violations," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 1–12.