2016 IEEE International Conference on Software Maintenance and Evolution

# Repairing intricate faults in code using machine learning and path exploration

Divya Gopinath Kaiyuan Wang Jinru Hua Sarfraz Khurshid
The University of Texas at Austin
{divyagopinath,kaiyuanw,lisahua,khurshid}@utexas.edu

***Abstract*—Debugging remains costly and tedious, especially for code that performs intricate operations that are conceptually complex to reason about. We present *MLR*, a novel approach for repairing faults in such operations, specifically in the context of complex data structures. Our focus is on faults in conditional statements. Our insight is that an integrated approach based on machine learning and systematic path exploration can provide effective repairs. *MLR* mines the data-spectra of the passing and failing executions of conditional branches to prune the search space for repair and generate patches that are likely valid beyond the existing test-suite. We apply *MLR* to repair faults in small but complex data structure subjects to demonstrate its efficacy. Experimental results show that *MLR* has the potential to repair this fault class more effectively than state-of-the-art repair tools.**

***Keywords*- program repair, semi-supervised learning, decision-tree learning, JPF, data-structures, condition faults.**

## I. Introduction

Debugging is usually a tedious and error-prone process. Recent years have seen the introduction of several novel techniques that automate the two key tasks in debugging: (1) identifying the faulty lines of code, termed *fault localization*; and modifying the faulty code to fix the faults, termed *program repair*, which is the focus of our work. A number of program repair techniques, e.g., those based on evolutionary algorithms [21], applying mutations to suspicious statements [5], performing code transformations [6], and leveraging repair templates [13], have been demonstrated to hold potential. However, performing intricate repairs involving code that operates on complex data structures remains a challenge. The large space of possible program variants impedes the efficiency and scalability of repair. Further, the accuracy of the repair often is not generalizable beyond the given test-suite [13], [17] or requires the presence of precise specifications [10].

```
    // pre-condition: list.size is correct
    public static boolean acyclicOrCircular(List list) {
1       List.Node l = list.header;
2       int cnt1 = 0;
3       boolean res = true;
4       while ((cnt1 < list.size) && (l!=null)) {
5           if (l == l.next) { //Error List1
5   //fix: if ((l.next!=null) && (l!=l.next.prev)){
6               res= false;
7               break;
8           }
9          l = l.next;
10          cnt1++;
11      }
12      return res;
13  }
```

Listing 1. List.repOK

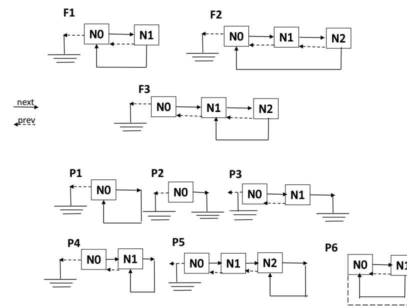


Fig. 1. Example test suite. Tests $F1, \ldots, F3$ fail. Tests $P1, \ldots, P6$ pass. In each test, node $N0$ is the header of the list.

We present an approach , *MLR* (Machine Learning based Repair), that applies machine learning in tandem with systematic path exploration to produce more accurate repairs, even on structurally complex data, while using just the given test suite as the oracle for correctness. Our focus is on failures due to wrong behavior caused by a faulty condition (e.g., in an `if` or `while` statement) in the program. Recent studies have shown that bugs in branch conditions of imperative programs are quite common ( [15], [18]). Our *key insight* is that mining the data-spectra of the passing and failing executions of the conditional branches can help prune the search space for repair and generate patches that are likely valid beyond the existing test-suite.

### A. Motivating example

Listing 1 shows a typical example of a data-structure method, `acyclicOrCircular`, of a doubly-linked list data-structure(DLL). The list has two fields: `header` that points to the first node and `size` that is the number of nodes in the list. Each node has two fields, `next` and `prev`, that define the doubly-linked structure. The method is expected to return true only if the structure of the list is either *acyclic* or *circular* (i.e., has a cycle that contains all nodes), and false otherwise. The given implementation of the method fails to do so. The method only detects self-loops (`l == l.next`) as cycles, but lists with loops involving two or more nodes, such as `l == l.next.next`, are considered acyclic by the method. Further, the check is only performed w.r.t the `next` field and cycles involving the `prev` field go undetected. Executing the sample test-suite (Figure 1) on the faulty method leads to the failure of three tests (F1, F2, F3) where the lists have cycles other

978-1-5090-3806-0/16 $31.00 © 2016 IEEE
DOI 10.1109/ICSME.2016.75



IEEE computer society

than self-loops. It could be determined using preliminary code analysis or with the help of fault-localization techniques ( [11]) that the branch on line 5 (`if (l == l.next)`) is faulty.

**Challenges of existing approaches.**

Even if the exact location of the fault is known, it is challenging to accurately change the code to function as desired, specifically for programs dealing with data-structures with complex structural properties. Most automated repair approaches for data-structures rely on the presence of user-defined formal specifications of correctness [10], [20]. Typical techniques for repair perform search in the space of syntactic transformations of the program [21]. The search space for data-structure programs is specifically huge since at any program point (line 5 in Listing 1), in addition to the stack variables (`l`), there is typically a large heap coverage (reachable by repeated de-referencing of object pointers; `l.next.next`, `l.prev`, `l.next.prev`). It is not feasible to generate non-trivial repairs by the application of simple mutations to the existing expressions [6].

It is difficult to repair faults in branch statements accurately, specifically in recursive implementations of data-structures. This is because even a single execution of the program potentially has multiple invocations of the branch. Therefore, the space of possible behaviors of the branch is huge. The challenge is to determine the ideal branch behavior and the condition expression that not only makes the given set of tests pass but functions as expected on all possible inputs.

SemFix [17], introduced the idea of *oracle-based repair* to repair condition faults; (1) oracle-mining: encodes correctness constraints at the point of fault, (2) repair-synthesis: generates a patch that satisfies the constraints. NOPOL [7], which specifically focuses on condition faults, has been shown to be effective for repairs involving small changes to basic integer operations and has not been evaluated on data-structure programs. Staged Program Repair (SPR) [13] employs a heuristics-based express algorithm to determine the correct truth sequence for conditions and subsequently generates a patch based on fixed templates. Although SPR can handle number of defects in benchmark programs, most of the patches are *plausible repairs* not generalizable beyond the given tests.

### B. Machine learning for more effective repair.

There has been a surge in the application of machine learning and data-mining techniques in the field of software analysis and debugging ( [4], [8], [19], [14], [9]). Our technique *MLR*, akin to oracle-based repair, adopts the two-stage approach to repair condition faults; *correct behavior determination* and *correct condition generation*. However, *the novelty of our approach* is to leverage learning techniques to improve the efficiency and accuracy of the two stages of repair for condition faults in imperative programs. The key steps of *MLR* are as follows;

**(1) Ideal behavior prediction using semi-supervised learning.** We employ *semi-supervised learning enabled by support vector machines, SVM* [3], to predict the correct behaviors of the faulty branch on the failing executions based on its behavior on similar passing executions.

Our *insight* is that for faults in conditions, there is useful information *latent* in the *data-spectra* of tests, i.e. distribution of data in the state space of passing and failing executions, which could be leveraged to determine the correct branching behavior and the correct conditional expression. The rationale being that a branch condition is likely to behave in the same manner on two different invocations if they display similar input state at that particular program point. Therefore we hypothesize that we can guess the correct behaviors for a branch in failing test executions based on its behaviors in passing test executions with similar state at that program point. This forms a typical setting for *semi-supervised learning* techniques such as **Support Vector Machines**, wherein the *learner* predicts the labels for a set of unlabeled data-points based on their similarity to labeled data-points.

(2) **Patching inaccuracies using systematic path exploration.** The accuracy of SVM based prediction is typically high for linearly separable data. The behavior of branch conditions need not conform to this. Therefore, there may be cases where the predicted behavior is incorrect. It is difficult to predict what type of kernels (polynomial, exponential) might give accurate classification for the problem. Instead, we perform *systematic exploration of the paths through the branch* (enabled by **Java PathFinder (JPF)** [2]) to determine the correct behaviors. We take guidance from the predicted behavior to prune the search space for path exploration.

(3) **Correct condition generation using decision-tree learning.** We synthesize the condition expression that would produce the expected behavior for both the failing and passing tests using *decision-tree learning* [16].

The key insight driving this idea is that an imperative branch condition can be considered as a *classifier* which classifies a given set of data into respective classes, based on the values of their attributes. A branch condition could be considered as a 2-way classifier with class labels +1 and -1, and the state variables acting as the attributes. **Decision-tree learning**( [16]) learns the *most compact* classifier for a given set of labeled data, as disjunctions of conjunctions of conditions on the data attributes. Heuristically, for a given set of labeled data, the smallest condition that can classify the data accurately tends to be the most generalizable. We hypothesize that it is highly probable that a compact decision-tree built using the inputs encountered in the given tests would be equivalent to the globally valid repair.

*MLR* generates the following alternate condition for the branch on line 5, $if((l.next! = null) \&\& (l! = l.next.prev))$. This condition correctly describes the intended behavior of the condition on Line 5 that not only works for the given tests but any DLL with correct input size. Neither SPR nor GenProg [21] could generate the correct repair for this faulty program given the same test suite (Section III).

### C. Contributions

We make the following contributions:

- ***MLR* leverages learning to mine the data-spectra of passing and failing executions to improve the effectiveness of oracle-based repair**.

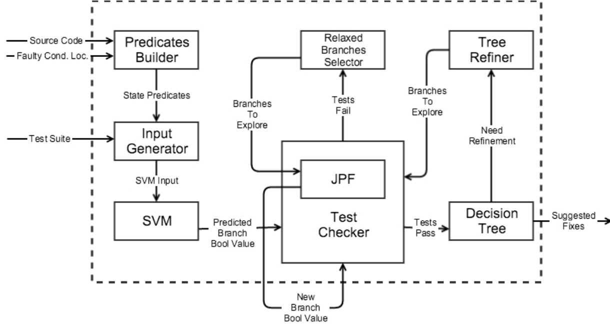


Fig. 2. *MLR* architecture.

- We present the first application of **semi-supervised learning to predict the ideal behavior of faulty branch** conditions and employ systematic path exploration to patch inaccuracies in the predicted behaviors.
- We present a novel idea to **learn branch conditions as compact decision-trees** which have high likelihood of being generalizable beyond the given test-suite.
- Our experiments demonstrate the usefulness of *MLR* to perform **intricate repairs accurately** and its ability to correct some **condition faults more accurately than state-of-the-art** techniques on our data structure subjects.

## II. Approach

This sections describes *MLR*'s algorithm. The inputs to *MLR* are: (1) the faulty program (`p`), (2) the impacted branch statement (`b`); we assume that the only impact of the fault is on a single branch statement and that a mechanism to localize the fault is available [11]; and (3) a test suite with failing tests (`f`) and passing tests (`p`); we assume that the impacted branch statement has some passing test coverage.

### A. Ideal behavior prediction using SVM

A branch condition in an imperative program could be represented as $(b_0 \; op_0 \; b_1 \; ... \; op_{n-1} \; b_n)$ where $b_i$ is a boolean predicate and $op_i$ is either a $\wedge$ or $\vee$ operator. Each boolean predicate is typically a condition in terms of the program variables, i.e. a query on the state of the program at that program point. Every invocation or execution of the branch $e_i$ could be characterized in terms of the valuations of the program state variables at that program point and its corresponding behavior or boolean branch output $o_i$, $e_i :=< \{var_0^i, ..., var_v^i\}, o_i >$. The branch is executed by `f` failing tests and `p` passing tests. There could be multiple invocations of the branch in a single test. We could consider the set of all branch invocations in all the tests and their respective outputs as a table,

$F_1$:$\{< \{var_0^0, ..., var_v^0\}, o_0 >, ..., < \{var_0^n, ..., var_v^n\}, o_n >\}$,... $F_f$:$\{< \{var_0^0, ..., var_v^0\}, o_0 >, ..., < \{var_0^m, ..., var_v^m\}, o_m >\}$ and $P_1$:$\{< \{var_0^0, ..., var_v^0\}, o_0 >, ..., < \{var_0^{n`}, ..., var_v^{n`}\}, o_{n`} >\}$, ... $P_p$:$\{< \{var_0^0, ..., var_v^0\}, o_0 >, ..., < \{var_0^{m`}, ..., var_v^{m`}\}, o_{m`} >\}$

where $n, m, n`, m`$ represent the number of invocations of the branch in the respective failing and passing tests, $o_i$ is the branch output encountered during the execution of the tests.

Support Vector Machines take as input a set of data-points with binary labels, some labeled and some unlabeled, each characterized by a fixed set of features. The above table is fed as input to the SVM. Every invocation of the branch is one row of input, characterized by the state variables as the features. The output of the branch in the original execution corresponds to the class values for passing tests (+1 for true, -1 for false), the class values are set to 0 (unknown) for failing executions. SVM predicts the labels of the unlabeled points based on their spatial proximity or geometric distance (calculated based on the feature values) to the labeled data-points. It learns the model of the classifier as a hyperplane that divides the data space into two groups containing data-points with positive and negative labels respectively. The unlabeled points are predicted to have +/- labels based on the side of the hyperplane they lie on. The data-points which are in close proximity to known labels are assigned high confidence of having correct predictions while those that lie near the margin are assigned low confidence values.

We heuristically select a set of *relevant state variables* to act as the features for SVM; (1) variables that get defined and/or updated within two edges of the faulty branch statement in the control flow graph, (2) expressions formed by de-referencing of every field in user-defined type variables, and (3) predicates comparing object pointers with $null$ and other pointers of same type.

Applying SVM to the three failing tests in our example, we found that labeling based on predictions made all the tests pass. The branch condition is predicted to evaluate to true whenever it detects a cycle while parsing the list inside the `while` loop ($2^{nd}$ invocation of the branch for F1, and $3^{rd}$ invocations for F2 and F3 respectively).

### B. Patching inaccuracies using systematic path exploration:

In case the predicted behavior does not make the tests pass, we adopt the following iterative algorithm to progressively relax the predicted labels on certain execution states and rerun the tests until they pass. We relax the predicted branch outputs on execution states where the confidence of prediction is below a certain threshold. We then rerun the test and use JPF to non-deterministically select the behavior of the branch on these relaxed execution states. This process is repeated until the test passes. The threshold is set to the minimum confidence value of all predictions for that test in the first iteration and then progressively updated to the second minimum one, so on.

### C. Correct condition generation:

Once we obtain the correct behavior of the branch for all the tests, the updated execution table, containing the execution rows and the corresponding outputs of the branch for all the tests, becomes the input to the decision-tree learner. A home-grown implementation of the ID3 algorithm [16] is applied to generate the classifier for class +1 or the condition of the branch to evaluate to true.

ID3 algorithm builds the decision-tree by recursively splitting the given set of data in such a way that each sub-group has a maximum of one type of class/label. It does so by determining a test or a predicate on an attribute that provides the maximum reduction in the entropy of the dataset or maximum information gain. The classifier thus generated is in the form of disjunction of conjunctions of attribute

TABLE I
*MLR* REPAIR RESULTS

| Err | Buggy Version | Generated Repair | Oracular Fix |
|---|---|---|---|
| LIST1 | if (l == l.next){...} | if ((l.next != null) && (l != l.next.prev)){...} | ✓ |
| RBT1 | while((p != null) && (p.left == null)) | while((p != null) && (ch == p.right)) | ✓ |
| RBT2 | while(p != null) | while((p != null) && (ch == p.left)) | × |
| RBT3 | ch = e.parent.right | ch = e.parent.right ... if ( e == p.right ) | ✓ |
| RBT4 | if (e.right != null) && (e.left != null) | if (e.right != null)* | ✓ |
| BST1 | while((x != null) && (k < x.key)) | if ((x != null) && (k < x.key))* | ✓ |

Oracluar Fix: ✓represents that generated repair is semantically equivalent to the oracular fix. Red shows deleted code in repair. Blue shows added code in repair. * represents repairs with heuristic refinement.

predicates. The classifier is 100% accurate on the input data since it has 0% error threshold. The decision-tree thus learnt for our example is $((l.next! = null)\&\&(l! = l.next.prev))$, which is the correct condition to detect cycles in a DLL.

We perform heuristic refinement of the generated classifier, as a post-processing step, to prevent over-fitting to the given test-suite. The algorithm heuristically patches the generated condition such that the difference with the already existing condition is minimized while still maintaining accuracy over the test-suite. Similarly, there could be more than one possible correct labelings that make the failing tests pass and therefore more than one classifiers generated. The algorithm ranks the classifiers based on a metric of *compactness* and prunes out solutions with more than a threshold number of predicates.

## III. PRELIMINARY EVALUATION

The purpose of our preliminary evaluation of *MLR* was – (1) **RQ1:** To test our hypothesis about the efficacy of learning techniques in improving the accuracy and efficiency of repair for condition faults; and (2) **RQ2:** Assess the contribution of the approach in terms of being able to correct this class of faults more effectively than the state-of-the-art.

We evaluate our approach on the following data structure methods with complex functionalities and properties; Doubly-linked list's `List.acyclicOrCircular`, Binary search tree's `insert` and Red-black tree's `containsValue`. For each subject, we create a test suite with full statement coverage and branch coverage. We seeded faults in branch conditions to simulate different types of error scenarios (Table I). Our implementation uses SVM Light [3] in transductive learning mode, JPF version 6.0, and a home-grown implementation of the ID3 decision-tree learning algorithm. All experiments were conducted in a 2.53Ghz CPU, 4GB RAM laptop running Windows 7.0.

**RQ1: Efficacy of using learning in correct behavior prediction and condition generation.** We evaluated the effectiveness of SVM based behavior prediction to accurately predict the ideal behavior of the faulty branch for the failing tests; *Prediction Accuracy* (Table II) calculated as the % of *relevant* predictions that were accurate. *Relevant predictions* refer to predictions for those branch invocations whose input

TABLE II
METRICS FROM SVM-BASED PREDICTIONS IN *MLR*

| Err | Passing test coverage(%) | Predictions based labeling | | Reduction in JPF search space(%) |
|---|---|---|---|---|
| | | Recall(%) | Precision(%) | |
| LIST1 | 53(9/17) | 100 | 100 | 100 |
| RBT1 | 60(12/20) | 53(8/15) | 87.5(7/8) | 91.6 |
| RBT2 | 18.5(5/27) | 60(12/20) | 41.6(5/12) | 3.6 |
| RBT3 | 40(13/32) | 42(8/19) | 62.5(5/8) | 94.2 |
| RBT4 | 45(14/31) | 100(7/7) | 57(4/7) | 81 |
| BST1 | 68.4(13/19) | 50(4/8) | 50(2/4) | 25 |

TABLE III
COMPARISON OF *MLR* WITH GENPROG AND SPR

| Err | *MLR* | GenProg | | | | SPR | |
|---|---|---|---|---|---|---|---|
| | | $W Loc$ | | $W_{out} Loc$ | | $W Loc$ | $W_{out} Loc$ |
| | | Brute | GA | Brute | GA | | |
| LIST1 | ✓ | × | × | × | × | × | × |
| RBT1 | ✓ | × | × | — | — | × | — |
| RBT2 | — | × | × | × | × | × | — |
| RBT3 | ✓ | × | × | × | — | × | × |
| RBT4 | ✓ | × | × | — | — | ✓ | ✓ |
| BST1 | ✓ | × | × | × | × | × | — |

✓represents correct fix, — represents plausible fix, × represents not generating any fix. $W Loc$ represents providing bug location, $W_{out} Loc$ represents providing the entire search space. Brute represents Brute search strategy, GA represents Genetic Algorithm search strategy.

state-vectors were present in the original execution and were also encountered during the final correct execution of the corresponding test (measured by *Recall*).

We observed that the prediction accuracy is highly sensitive to the passing test coverage, behavioral diversity and the accuracy of the original branch condition on all invocations of the passing test executions. This is expected since the passing executions form the basis for the predictions. Low recall also leads to poor prediction accuracy since the data points used for learning are not representative enough. Despite the inconsistency in the prediction accuracy, we found that predictions-guided pruning of search space helped reduce the space for path exploration by JPF by more than 80% in 4 cases.

Finally, we investigate the accuracy of the condition generation by Decision-Tree learning. The generated repairs were 100% accurate for the given tests. The repairs were also equivalent to the ideal repair in all cases except one (RBT1), which was overfitted due to the incompleteness of the tests in the suite to cover all possible inputs impacting the behavior of the branch.

**RQ2: Repairability.** We ran the state-of-the-art repair techniques, SPR and GenProg, on the same examples with the same seeded faults and test-suites. Table III compares our approach (*MLR*) with these two techniques. Given the *location of the fault*, *MLR* is able to generate fixes for all the bugs; 5 correct and 1 plausible. GenProg is unable to generate fixes for any of the bugs. This could be because the fix ingredients required for these patches are not be found in any other part of the code. SPR is able to generate a fix (correct) only for RBT4. For RBT1 and RBT2, it does determine a behavior for

the faulty branch, that would make the tests in the suite pass, but is unable to find an expression that produces that behavior. When SPR and GenProg are allowed to modify any part of the code, they generate fixes for more number of cases. GenProg generates a patch for RBT1, RBT3 and RBT4 [1] that inserts multiple lines of code. These patches make the existing tests pass, however they alter the ideal functionality of the code. SPR generates the correct fix for RBT4 and *plausible* fixes for RBT1, RBT2 and BST1 [1]. For instance, for RBT1 the patch bypasses the faulty $while$ statement unconditionally, which makes the tests in the suite pass but is not the ideal repair. Our approach is able to determine the ideal behavior and the ideal repair expression based on the behavior of the branch on passing executions.

*Threats to Validity and Future Extensions.* We recognize that the following factors impact the generalizability of the results, (1) the number and size of the subject programs are small, and they are all data structure programs; (2) the faults are manually seeded in conditions and the fault location is known precisely; and (3) the tests are manually created. An empirical quantification of the minimum passing test coverage required to obtain near accurate results could act as a guide to users of the tool. Further, heuristic selection and prioritization of state-predicates such as based on matching with other parts of the code, occurrence of local variables and number of de-references in the predicate expressions, could aid in preventing overfitting of generated repair to the tests.

## IV. Related Work.

This section compares our work with recent repair approaches using machine learning.

Our previous work [9] employs machine learning to repair incorrect *Where clauses* in database statements in ABAP programs. In this paper, we have extended the idea to repair branch conditions in imperative programs. Although the crux of the approach is similar, there are considerable differences between the two domains. Input to database statements are relational tables with rows characterised by fields, which is exactly the form of input for SVM. The behavior of the selection for each row is independent of the others. In an imperative program, on the other hand, the mapping of program state to SVM input format is not straight-forward. Each row represents different occurrences of the branch condition during a single execution and hence they may be dependent on each other. This necessitates the use of JPF to explicitly execute the code to explore subsequent states. The output of a selection statement is always a subset of its input rows and a SAT-based combinatorial search is employed in the previous work.

Recently proposed approaches ( [14], [12]) employ learning to mine patches from other bug-fixes. They differ from our approach in the context in which they uses learning. Prophet [14] learns probabilistic models of correct code from a set of successful patches in open source software repositories and applies these models to rank patches to correct a given faulty program. The application of learning is after the generation of a population of candidate patches to rank the possible patches. In *MLR* we apply learning to prune the search space and build more accurate patches at the generation stage itself.

## V. Conclusion

We introduced *MLR*, a novel approach for repairing intricate condition faults in data structures programs. *MLR* mines the data-spectra of the passing and failing executions of the faulty branch to prune the search space for repair and generate globally-valid patches. We believe *MLR* provides a viable foundation for effective repair of faulty conditions which could be integrated with existing more generic routines in the future.

## VI. Acknowledgments

This work was funded in part by the National Science Foundation (NSF Grant No. CCF-1319688).

## References

[1] Comparison with genprog and spr. https://github.com/lisahua/ICSME16-comparison/. Accessed: 2016-04-10.
[2] Java path finder. http://babelfish.arc.nasa.gov/trac/jpf.
[3] Svm. http://svmlight.joachims.org/.
[4] D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel. Automated support for diagnosis and repair. *Commun. ACM*, 58(2):65–72, 2015.
[5] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *ICSE*, pages 121–130, 2011.
[6] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *ICST*, pages 65–74, 2010.
[7] F. Demarco, J. Xuan, D. L. Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with SMT. In *In 6th International Workshop on Constraints in Software Testing, Verification, and Analysis, CSTVA*, pages 30–39, 2014.
[8] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. C. Rinard. Inference and enforcement of data structure consistency specifications. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 233–244, 2006.
[9] D. Gopinath, S. Khurshid, D. Saha, and S. Chandra. Data-guided repair of selection statements. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 243–253, 2014.
[10] D. Gopinath, M. Z. Malik, and S. Khurshid. Specification-based program repair using SAT. In *TACAS*, pages 173–188, Mar. 2011.
[11] J. A. Jones. *Semi-Automatic Fault Localization*. PhD thesis, Georgia Institute of Technology, 2008.
[12] X. D. Le, D. Lo, and C. Le Goues. History driven program repair. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016*, 2016.
[13] F. Long and M. Rinard. Staged program repair with condition synthesis. In *10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, pages 166–178, 2015.
[14] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 298–312, 2016.
[15] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.
[16] T. Mitchell. *Machine Learning*. McGraw Hill, 1997.
[17] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: program repair via semantic analysis. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 772–781, 2013.
[18] K. Pan, S. Kim, and E. J. W. Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.
[19] R. Sharma, A. V. Nori, and A. Aiken. Interpolants as classifiers. 2012.
[20] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *ISSTA*, 2010.
[21] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–374, 2009.