Common Statement Kind Changes to Inform Automatic Program Repair

Mauricio Soto
Carnegie Mellon University
Pittsburgh, PA
mauriciosoto@cmu.edu

Claire Le Goues
Carnegie Mellon University
Pittsburgh, PA
clegoues@cs.cmu.edu

ABSTRACT

The search space for automatic program repair approaches is vast and the search for mechanisms to help restrict this search are increasing. We make a granular analysis based on statement kinds to find which statements are more likely to be modified than others when fixing an error. We construct a corpus for analysis by delimiting debugging regions in the provided dataset and recursively analyze the differences between the Simplified Syntax Trees associated with EditEvent's. We build a distribution of statement kinds with their corresponding likelihood of being modified and we validate the usage of this distribution to guide the statement selection. We then build association rules with different confidence thresholds to describe statement kinds commonly modified together for multi-edit patch creation. Finally we evaluate association rule coverage over a held out test set and find that when using a 95% confidence threshold we can create less and more accurate rules that fully cover 93.8% of the testing instances.

CCS CONCEPTS

• **Software and its engineering** → Software evolution;

ACM Reference Format:
Mauricio Soto and Claire Le Goues. 2018. Common Statement Kind Changes to Inform Automatic Program Repair. In *MSR '18: MSR '18: 15th International Conference on Mining Software Repositories , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3196398.3196472

1 INTRODUCTION

Bug repair is one of the most resource consuming tasks in software development [2, 12, 15]. In the last decade there has been increasing attention to techniques for automatic program repair (APR), which repair software bugs automatically (e.g. [5, 6, 13, 14]). One family of techniques follows a syntactic *generate-and-validate* approach (e.g. [3, 5, 10, 13]). These techniques take as input a buggy program and a test suite containing passing tests cases and failing test cases (which expose a defect). The approaches then *generate* patch candidates by applying and combining syntactic source transformations, and *validate* these candidates against the test suite.

MSR '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196472

One challenge in this process is that the search space of patch candidates is trivially infinite. APR approaches thus restrict the considered transformations, and use off-the-shelf fault localization [13] to reduce the number of locations considered for editing. One problem is that statistical fault localization can often assign multiple statements the same score [4]. Even further, patches that require more than one edit are commonly found in the real world and have been vastly understudied. The need for mechanisms that help guide the search space for multi-edit patches keeps growing.

Evidence shows that not all statements are modified equally, and there are benefits from using history-based data [5, 11]. Therefore, we first analyze the most commonly modified statement kinds and evaluate how the fault localization process can benefit from assigning a higher priority to the statements most commonly modified. Second, we create association rules to describe relationships between statement kinds commonly modified together to inform the creation of multi-edit patch candidates.

The MSR Challenge dataset [9] describes the actions performed by developers when coding. We mined changes performed in the debugging process by delimiting debugging regions and comparing the simplified syntax trees (SST) of consecutive *EditEvent's* using an state-of-the-art tree differencing tool [8]. We then apply the association rule learning algorithm Apriori [1] to this corpus to obtain relationships that describe what statement kinds are edited commonly together.

We evaluate the level to which the rules can predict edits by analyzing what sections of the events are covered by the learned association rules. Finally we analyze the confidence level of the learned rules. We found that when we use a 95% confidence threshold, we can create association rules that maintain the same flexibility as lower confidence thresholds, but increases the accuracy of the created rules and decreases the number of rules delimiting even more the search space of potential edits performed by APR approaches to obtain successful patches.

The main contributions of this paper are the following:

- We **describe the distribution** of statement kinds being modified by developers when fixing errors in source code.
- We **evaluate the selection process** of statement kinds to be modified to fix source code errors, using history-based data (49.54% correctly guessed), compared against an equally distributed random approach (5.11% correctly guessed).
- We **create association rules** that describe the behavior of statement kinds that developers modify commonly together. We evaluate the expressive power of the learned rules, we find that we can fully cover 93.8% of the held-out events while maintaining 95% confidence.

2 CREATING THE CORPUS

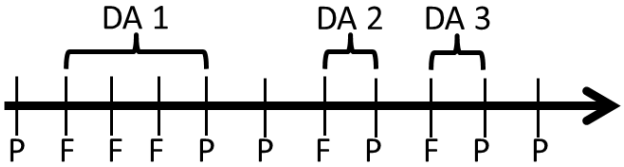
We collect a corpus of events based in modifications developers perform when fixing an error from the MSR Challenge dataset [9]. We then use this corpus to find differences between consecutive events and their corresponding Simplified Syntax Tree's (a snapshot of the source code when the event took place) to create a distribution of statement kinds modified and association rules.

2.1 Delimiting Debugging Regions

The first step towards analyzing the debugging process is identifying periods during which a developer is trying to fix a bug. We delimit the debugging regions by starting at a point in time δ where a *TestRunEvent* is triggered and all test cases were run (test case execution was not aborted) but one or more test cases failed, while in the previous *TestRunEvent*, all tests passed. The delimitation ends at a later point in time ϵ where ϵ is the closest *TestRunEvent* later in chronological order where all test cases were ran and passed.

Figure 1 shows an example of a time line with only the *TestRunEvent*s shown. "P" represents a case where all test cases passed; "F" represents a case where one or more test cases failed. The `debugging areas` are shown by the brackets demarcated with "DA". Note that between delimiting points δ and ϵ there can be several *TestRunEvent's* with failing test cases, meaning that the developer is actively debugging the error pointed out by the failing test cases, but still has not been able to make all test cases pass.

Figure 1: Example of the process used to obtain the demarcation of debugging areas. Events where all test cases pass (P) and events where one or more tests fail (F) are analyzed to create debugging areas (DA).



We identified 634 debugging sequences in the dataset provided by the MSR Challenge [9]. Within these debugging sequences, there are 1,251,334 unique events, from which 1,748 are *EditEvent's* that have associated a not-unknown Contexts, and therefore contain a valid Simplified Syntax Tree. These *EditEvent's* are our primary subject of analysis.

2.2 Simplified Syntax Tree Differencing

Given two consecutive *EditEvent's* within a debugging region, we proceed to analyze the differences between their two associated Simplified Syntax Trees (SST). These correspond to the modifications the developer performed while debugging. We use the state-of-the-art tree differencing tool APTED [8], recursively converting the SST's into a representation readable by APTED. Once we know which nodes were modified between *EditEvent*s, we record their corresponding statement types.

2.3 Statement Kind Distribution and Association Rule Creation

We analyzed the distribution of statement kinds modified in the corpus. EditEvent's may modify more than one statement kind, therefore the sum of these distribution does not sum to 100%. From our 1,748 events, the most commonly modified statement kinds are `ExpressionStatement`'s, modified in 1,272 (72.8%) corpus events, followed by `Assignment`'s modified in 1,186 events (67.8%). `VariableDeclaration`'s are modified in 1,071 events (61.3%) and `ReturnStatement`'s in 714 events (40.8%).[1] This confirms our initial hypothesis that not all statement kinds are modified equally.

To collect rules for statement kinds that are modified together in multi-edit patches, we use Apriori [1] to create association rules. Association rule learning is a machine learning mechanism that identifies relationships between objects in large datasets. Association rules are implications of the form $X \implies Y$. This learning mechanism takes into account *Confidence* (a measure of the likelihood of the consequent section being present given the antecedent) and *Support* (how frequently the set of statement types in the rule occurs in the corpus). Using our corpus of edits and their associated statement types, we create association rules that provide guidelines for what statement types are modified together commonly.

3 EVALUATION

We evaluate the modified statement kind selection process by comparing a random selection against our History-based approach which gives higher priority to more commonly modified statement kinds. We then evaluate the association rules by measuring the percentage of edit instances they correctly predict, or cover.

3.1 History-Based Statement Selection

To evaluate the distribution of the statement kinds modified in the corpus, we compare against an equally distributed counterpart. This simulates the process being used by most current APR approaches where code is analyzed, using a fault localization mechanism, and a statement is randomly selected from the most suspicious block.

We evaluate the process of selecting a statement kind informed by history-based data and check if it correctly predicts held-out instances. We use 10 fold cross validation to avoid training and testing on the same data. For each fold, we iterate through each EditEvent. For the equally distributed approach, we randomly select one from all possible statement kinds and check if the selected statement kind was modified in the EditEvent. For our history-informed approach, we create a distribution of the statement kinds modified from the complement of the fold, and select a statement kind bounded by the associated distribution. The more commonly modified statement kinds are more likely to be chosen than the less commonly modified statement kinds.

Table 1 describes the results per each evaluated fold. On average, the History-based approach is able to correctly guess one of the statement kinds 49.54%. The Equally distributed approach on average is only able to correctly guess 5.11% of the time. Two sample t-test indicates that the difference between the Equally distributed and History-based approaches is statistically significant ($\alpha < 0.05$).

[1] The full list is available in https://github.com/squaresLab/MSRChallenge2018

Table 1: Evaluation of modified statement kind selection (Equally distributed vs History-based).

	Equally Distributed		History-Based	
Fold	Count	Percentage	Count	Percentage
1	13	(7.39%)	72	(40.90%)
2	4	(2.27%)	98	(55.68%)
3	11	(6.25%)	86	(48.86%)
4	7	(3.98%)	84	(47.72%)
5	10	(5.68%)	92	(52.27%)
6	7	(3.98%)	89	(50.57%)
7	10	(5.68%)	92	(52.27%)
8	8	(4.54%)	89	(50.57%)
9	11	(6.25%)	96	(54.54%)
10	9	(5.11%)	74	(42.04%)
Mean	9	5.11	87.2	49.54
Std Dev	2.45	1.39	8.15	4.43

3.2 Association Rule Coverage

We next evaluate the effectiveness of the association rules and their potential for use in building the necessary edits to create successful multi-edit patches. We first remove from our corpus the instances where the developers performed a single edit. This removed 578 instances from the initial 1,748. We perform this restriction because association rules by definition require at least two elements: an antecedent and a consequent.

We then analyze how often the association rules can build the modifications in the events by using rule coverage. We use 10 fold cross validation to avoid training and testing in overlapping data. First, we divide our corpus into 10 folds. For each of the folds we perform the following actions: the selected fold is used as testing data, and the remaining nine folds are used to create association rules. We then analyze how many of the events in the testing data can be built from the association rules created from the remaining nine folds. The learned rules can either cover all the edits in an event (the event is *fully covered*), they can cover some of the edits in an event (the event is *partially covered*), or they can cover none of the edits in an event (the event is *not covered*).

We show an example for further understanding of the coverage concept. Table 2 shows three events. In our example Event 1 describes an event where the developer modified an `Assignment`, a `ForLoop` and an `IfElseBlock`. The rules shown in Table 2 represent rules created from the remaining nine folds of the corpus. In our example: Rule 1 indicates that when an `Assignment` and a `ForLoop` have been modified, the next step is to modify an `IfElseBlock`.

In this example, Event 1 is fully covered since Rule 1 covers all the edits in the event. For Event 2, Rule 1 does not apply, even though Event 2 contains the antecedent of Rule 1, it does not contain the consequent, therefore the rule is discarded. Rule 2 does apply to cover the latter two edits of Event 2, therefore Event 2 is partially covered. Finally Event 3 is not covered, even when some of its edits are contained in the antecedent of the rules, there is no rule that would successfully predict the behavior of this event.

Table 2: Events (top); learned association rules (bottom).

	Events
1	Assignment; ForLoop; IfElseBlock
2	Assignment; ForLoop; VariableDeclaration
3	Assignment; ForLoop; ReturnStmt
	Association Rules
1	Assignment ∧ ForLoop → IfElseBlock
2	ForLoop → VariableDeclaration

Finally we performed this analysis at 3 different confidence thresholds (90%, 95%, 100%) to compare the accuracy of the rules against the number of rules that can be created with lower confidence. Higher confidence produces a small set of very accurate rules, in which it is very likely that when the antecedent is present the consequent will be present as well. But this approach does not allow for much flexibility and the rules overfit to the corpus they were created from. Lower confidence creates a wide set of rules that are less accurate (if the antecedent is present is does not necessarily mean that the consequent will be present as well) but more flexible.

For each confidence threshold of 90%, 95%, and 100%, we evaluated association rule coverage on a held out dataset, using a standard 10 fold cross validation process. Finally, we aggregate the results from all folds.

Figure 2 shows our results for each of the different confidence thresholds as described below.

90% confidence threshold: When evaluating the rules created under a 90% threshold, 94.14% of the events are fully covered, 1.87% are partially covered, and 3.99% are not covered. When using this approach, an average of 2837.2 rules were created for the 10 folds.

95% confidence threshold: 93.80% of the testing instances are fully covered, 2.04% are partially covered, and 4.9% are not covered. On, average 2134.3 rules were created.

100% confidence threshold: Finally, 35.85% are fully covered, 2.29% are partially covered, and 72.80% are not covered. On average, 1484.6 rules were created when using this approach.

We found that when creating association rules with empirical data for automatic program repair, 95% confidence appears idea: it maintains very similar flexibility to the 90% threshold with improved accuracy. This also means that the number of rules created decreases, which helpfully restricts the search space by restricting it to a smaller number of more accurate rules.

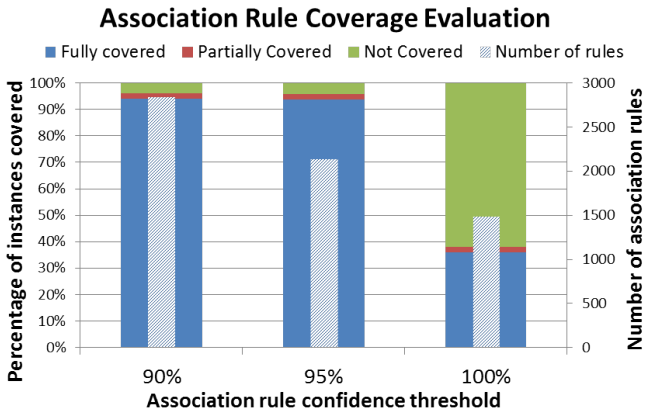
3.3 Threats to Validity

Internal validity: To tackle possible errors in our implementation and experiments, we release our code to be reviewed by the other researchers.[2] We use the debugging regions based on *TestRunEvent's* as a proxy for debugging intent since capturing developer's intent is a non-trivial task. We also use 10 fold cross validation to reduce the risk of training and testing on the same data.

External validity: It is possible that our results will not generalize to external datasets and to real bug fixes. To mitigate this concern,

[2] All instruments and source available in https://github.com/squaresLab/MSRChallenge2018

Figure 2: The wide bars are described by the left Y axis. This represents the percentage of instances covered fully, partially or not by the association rules (higher full coverage is better). The thin bars are described by the right Y axis. This represents the number of rules created for each confidence threshold (lower is better). The bars from left to right describe the cross validation results when using 90%, 95% and 100% confidence thresholds correspondingly.



we have created our corpus from the dataset made available to us by the MSR Challenge [9] which records the steps developers take while in the software development process. This dataset is gathered from a diverse pool of volunteers with different backgrounds and levels of expertise.

4 RELATED WORK

Soto et al. [11] have created association rules to inform the APR process basing their corpus in **mutation operators** taken from popular Github projects written in Java using commit level granularity, different from our approach which creates association rules for **statement types** based in finer grained C# code changes [9]. Mutation operators help to inform what *action* to perform next, while statement types help to inform what object to modify next. Par [5] describes an analysis of common changes applied by humans when fixing errors and templates that can be mined from their corpus. HDRepair [3] uses fix history at a broader level to assess patch suitability. Zhong and Su [16] conduct an empirical study on six popular Java projects analyzing the incidence of three mutation operators. Martinez and Monperrus [7] study mutation operator incidence across 14 projects. Prophet [6] creates a probabilistic model from the history of 8 different projects to rank candidate patches.

5 CONCLUSIONS

In this study we mined the dataset provided by the MSR Challenge [9] to create a corpus of highly granular edits performed by developers in the process of fixing an error in source code. We identify debugging regions and inspect the changes between the simplified syntax trees from each of the *EditEvent's* to create a corpus of events. We analyze the distribution of commonly modified statement kinds and evaluate how a search process bounded by this history can correctly guess (49.54% of cases) what statement kind is modified to fix an error, as opposed to its equally distributed counterpart (5.11% of the cases).

We then create association rules to provide guidelines of what kinds of statement to edit together to create successful multi-edit patches. We measure how many of the events can be fully, partially, or not covered by the association rules; we find that 95% appears to be the preferred confidence threshold. We are able to fully cover 93.80% of the events with a 95% confidence threshold which increases the accuracy of the rules created and decreases the number of rules, therefore helping delimit the vast search space in automatic program repair. These findings can be used to guide the creation of multi-edit patches in an APR context therefore delimiting the vast search space and providing guidelines for properly selecting statement kinds to modify based in the history of developer changes.

ACKNOWLEDGMENTS

This research was partially funded by AFRL (#FA8750-15-2-0075) and the NSF (CCF-1563797); the authors are grateful for their support. Any opinions, findings, or recommendations expressed by the authors do not necessarily reflect those of the US Government.

REFERENCES

[1] Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast Algorithms for Mining Association Rules in Large Databases. In *20th International Conference on Very Large Data Bases (VLDB'94)*. 478–499.
[2] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. 2013. *Reversible debugging software.* Technical Report.
[3] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*.
[4] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of Test Information to Assist Fault Localization. In *International Conference on Software Engineering (ICSE'02)*. 467–477.
[5] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering (ICSE'13)*. 802–811.
[6] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Symposium on Principles of Programming Languages (POPL '16)*. 298–312.
[7] Matias Martinez and Martin Monperrus. 2015. Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing. In *Empirical Software Engineering (ESE'15)*. 176–205.
[8] Mateusz Pawlik and Nikolaus Augsten. 2015. Efficient Computation of the Tree Edit Distance. *ACM Transactions on Database Systems (TODS)* 40. Issue 1.
[9] Sebastian Proksch, Sven Amann, and Sarah Nadi. 2018. Enriched Event Streams: A General Dataset for Empirical Studies on In-IDE Activities of Software Developers. In *Proceedings of the 15th Working Conference on Mining Software Repositories.*
[10] Yuhua Qi, Xiaoguang Mao, and Yan Lei. 2013. Efficient Automated Program Repair through Fault-Recorded Testing Prioritization. In *International Conference on Software Maintenance (ICSM'13)*. 180–189.
[11] Mauricio Soto and Claire Le Goues. 2018. Using a Probabilistic Model to Predict Bug Fixes. In *Software Analysis, Evolution, and Reengineering 2018.*
[12] Gregory Tassey. 2002. *The economic impacts of inadequate infrastructure for software testing.* Technical Report.
[13] Westley Weimer, Michael Dewey-Vogt, Claire Le Goues, and Stephanie Forrest. 2012. A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering (ICSE'12)*. 3–13.
[14] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results. In *IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. 356–âĂŞ366.
[15] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. 2007. How long will it take to fix this bug?. In *International Workshop on Mining Software Repositories (MSR'07)*. 1.
[16] Hao Zhong and Zhendong Su. 2015. An empirical study on real bug fixes. In *International Conference on Software Engineering (ICSE'15)*. 913–923.