An Empirical Study on Reducing Omission Errors in Practice

Jihun Park
Korea Advanced Institute of Science and Technology
Daejeon, Korea
jhpark@se.kaist.ac.kr

Miryung Kim
The University of Texas at Austin
Austin, TX USA
miryung@ece.utexas.edu

Doo-Hwan Bae
Korea Advanced Institute of Science and Technology
Daejeon, Korea
bae@se.kaist.ac.kr

ABSTRACT

Since studies based on mining software repositories sparked interests in the field of guiding software changes, many change recommendation techniques have been proposed to reduce omission errors. While these techniques only used existing software commit data sets to evaluate their effectiveness, we use the data set of supplementary patches which correct initial incomplete patches to investigate how much actual omission errors could be prevented in practice. We find that while a single trait is inadequate, combining multiple traits is limited as well for predicting supplementary change locations. Neither does a boosting approach improve accuracy significantly, nor filtering based on developer or package specific information necessarily improves the accuracy. Developers rarely repeat the same mistakes, making the potential value of history-based change prediction less promising. We share our skepticism that omission errors are hard to prevent in practice based on a systematic evaluation of a supplementary patch data set.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

Keywords

omission error; supplementary patch; mining version history

1. INTRODUCTION

About ten years ago, Zimmermann et al. and Ying et al. [16, 18, 19] sparked interests in the promise of guiding software changes based on version histories. Over the past decade, many change recommendation systems have been proposed to identify additional change locations to reduce omission errors. For example, *FixWizard* [8] identified additional change locations using cloning based similarity, and Hassan and Holt [2] investigated several change propagation heuristics, finding that historical change coupling is more accurate than structural dependency relationships for predicting co-changed program entities. In addition to the aforementioned approaches, many change recommendation systems [3, 4, 6, 7, 17] have been developed to find additional change locations based on version histories.

These approaches suggested ways to predict supplementary change locations; however, they evaluated their accuracy on existing software commit data only. They grouped commits into a set of transactions, and then predicted the remaining entities of a transaction based on a subset of the transaction. We use a *supplementary patch data set* to predict additional change locations for real-world omission errors. The supplementary patch data set was created in our prior work [9] to study the feasibility of reducing omission errors, where developers applied supplementary patches to correct or complete original incomplete bug fixes.

We investigate how the supplementary change locations can be predicted based on the initial change locations. To represent the relationship between them, we propose a new graph representation called *change relationship graph (CRG)*. The CRG uses packages, classes, and methods as graph nodes, and represents the relationship between graph nodes based on structural dependency, historical co-change, code clone, and name similarity relationships. We then develop a path generalization algorithm based on the CRG to find the frequently occurring relationship between the initial and supplementary change locations.

Based on a comprehensive study, we have determined that it is inherently challenging to predict supplementary change locations based on initial change locations and then we share our skepticism that reducing omission errors based on a systematic evaluation of a supplementary patch data set is very difficult in practice. Our comprehensive study on the supplementary patch data set finds that neither a specific relationship nor a pattern exists between the initial and supplementary change locations.

2. RELATED WORK

Zimmermann et al. [19] and Ying et al. [16] showed the early promise for guiding software changes based on version histories. They used association rule mining to identify additional change locations. They evaluated whether their tool could identify the other entities based on one entity of a transaction. Zimmermann et al.'s approach showed 33% of precision and 29% of recall on average. The precision and recall of Ying et al.'s approach were in the range of 30% to 50%, and 10% to 30%, respectively. Even though they evaluated their approaches according to whether the approach

ASE'14, September 15-19, 2014, Vasteras, Sweden.
Copyright 2014 ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2642956.

was able to guide software changes of the transaction data set during the implementation phase, they did not evaluate how the approach could prevent omission errors, based on a real-world supplementary patch data set. In addition, the overall accuracy was not high enough to be pragmatic; our investigation finds that the accuracy becomes even lower when the historical co-change pattern analysis is applied to a real-world supplementary patch data set.

Robillard [11] and Saul et al. [12] used structural dependency in order to make a suggestion set for program investigation. These two approaches showed that structural relationships can identify *relevant* program entities, but the portion of omission errors that structural dependency can identify was not evaluated using a real data set.

FixWizard of Nguyen et al. [8], identified a group of program entities (*code peer*) that should be changed together. To find the code peers, they identified methods that share similar object usages. Because only a few supplementary patches have a content similar to the initial changes [9], the tool cannot be generalized to real-world omission errors from the supplementary patch data.

Herzig and Zeller [3] mined cause-effect-chains from version histories, which are represented in CTL (Computation Tree Logic). The change coupling, which is generalized by CTL, can represent a causal relationship between two locations within a predefined time interval. They evaluated how well their approach could predict additional change locations. Our investigation results show that only a small portion of omission errors can be predicted using this kind of mining technique; the majority of the patterns between initial and supplementary change locations appear only once.

Hassan and Holt [2] assessed different kinds of change propagation heuristics, and then suggested a hybrid approach. The hybrid approach combined historical co-change pattern and code layout based methods, and it showed 49% average precision and 51% average recall.

Malik and Hassan [6] improved Hassan and Holt's work[2] using adaptive change heuristics based on the *best heuristic table*, showing the possibility that managing prior prediction information for each entity can improve the prediction accuracy. (64% precision and 78% recall) Their hybrid approaches showed high recall and precision at predicting co-changed entities; however, the evaluation was only conducted on a transaction data set. We also find that a boosting technique based on past prediction accuracy information does not significantly improve the prediction accuracy.

Park et al. [9] investigated the characteristics of supplementary changes and how omission errors can be reduced. They found that a significant portion of bugs required supplementary patches. They also found that structural dependency and code clone analysis were limited at predicting supplementary change locations. In this paper, our investigation based on a graph representation (CRG) includes not only a single relationship between the program entities, but also a combination of the relationships, a boosting approach based on the past prediction accuracy, package or developer filtering based predictions, and repeated patterns between initial and supplementary change locations.

3. BACKGROUND

3.1 Supplementary patch data set

The supplementary patch data set was created in our prior work [9] to study the characteristics of supplementary patches and how we could reduce omission errors. In this paper, we use the supplementary patch data set to study the feasibility of several prediction approaches for reducing real-world omission errors. The supplementary patch data set comprises initial patches and supplementary patches. An initial patch represents an initial fix attempt to fix a bug; the fix then turned out to be incomplete or incorrect later. Supplementary patches are applied later to complete or correct the initial patch. This data set and analysis source code are available on the first author's web page.[1]

We use three open source projects as study subjects—Eclipse JDT core, Eclipse SWT, and Equinox p2. We identify the supplementary patch data set in the same way as we did in our prior work [9]. We connect commit logs with bug IDs by parsing the logs considering every integer sequence as potential bug IDs. We then ignore if the number is out of a pre-defined range; the minimum value is 3000 (to ignore small numbers), and the maximum value is 214100, 259850, and 298700 corresponding to bug ID at 2007/12, 2008/12, and 2009/12 in the three projects, respectively.

After connecting commits with bug IDs, we categorize the bugs into two groups. (1) *Type I bugs* are the bugs that are fixed only once, and (2) *Type II bugs* are the bugs that are fixed more than once in the study period. We call the first fix of Type II bugs the initial patch; the subsequent fixes are supplementary patches.

We consider the bugs reported in 2002/01 to 2007/12, 2002/01 to 2008/12, and 2006/10 to 2010/01, to make sure that the bugs are completely resolved. Among 3803, 4673, and 1783 of bugs, we find that 23%, 26%, and 26% of them require supplementary patches in Eclipse JDT core, Eclipse SWT, and Equinox p2, respectively.

3.2 Building a change relationship graph

To express a path between the initial and supplementary change locations, we develop a graph representation called the Change Relationship Graph (CRG). The CRG uses packages, classes, and methods as graph nodes, and structural dependency, historical co-change, code-clone and name similarity relationships as graph edges.

The CRG is the first representation that allows reasoning about multiple traits of change relationship; the multiple traits include structural dependency, historical co-change, code clone, and name similarity relationships.

Identifying graph nodes. We define program entities, such as packages, classes, and methods as graph nodes. We extract structural information from each version of the program files using the PPA (Partial Program Analysis) tool [1]. We utilize the PPA tool to generate AST only from changed files, not from the whole program corresponding to each revision. We track the version history of each program entity to identify the added revisions and deleted revisions.

Identifying structural dependency relationship edges. We use containment, inheritance, and method invocation relationship edges as structural dependency relationship edges. They are created based on the AST, generated by the PPA tool.

Identifying historical co-change relationship edges. We create co-change relationship edges between method nodes that are changed within the same revision. By parsing patches corresponding to each revision, we identify co-changed method

[1]http://se.kaist.ac.kr/jhpark

nodes, and then we create a co-change edge between every pair of the methods.

Identifying code clone relationship edges. We create code clone relationship edges between method nodes that have similar content in their method body. The similar contents are identified using CCFinderX [5] with a minimum token size of 40. We identify code clones every 4000 revisions, and then we map the code clone pairs to method nodes by parsing patches to make a code clone relationship edge.

Identifying name similarity relationship edges. We create name similarity relationship edges between method nodes that have similar names. We use the same similarity measure as that used in UMLDiff [15]. Because calculating the name similarity between every method node takes too much time and makes the number of name similarity edges huge, we make a name similarity edge when the following conditions hold: 1) two method nodes belong to the same package. 2) the containing classes of two method nodes have name similarity larger than 0.5. 3) the names of two methods have name similarity larger than 0.7.

3.3 Evaluating a prediction method

We evaluate how accurately a prediction method predicts supplementary change locations based on initial change locations. Our measures for assessing a prediction method are its precision, recall, and feedback. **Precision** and **recall** are common accuracy metrics. Precision evaluates whether the suggestion set accurately predicts actual supplementary locations; recall evaluates whether the actual supplementary locations are covered by the predicted set. Where P represents the predicted suggestion set and S represents the actual supplementary change locations excluding the initial change locations, precision and recall are defined as follows: $precision = \frac{|P \cap S|}{|P|}$, $recall = \frac{|P \cap S|}{|S|}$. If one of P and S is an empty set, we do not count the prediction in the result for the calculation.

Because we disregard cases in which a prediction method suggests an empty set as a candidate of supplementary change, we should compensate to assess *what portion of initial changes can obtain at least one suggestion.* We use the **feedback** measure, introduced by Zimmermann et al. [19] to evaluate whether a prediction method or a rule can be generally used. Where the predicted suggestion set, $\{P_b^m\}$, is derived using a prediction method m for bug b, the feedback is defined as follows:

$$feedback = \frac{|\ \{b \in TypeIIbugs \mid 1 \leq |\{P_b^m\}|\ \}\ |}{|\ \{TypeIIbugs\}\ |}$$

The numerator represents the number of Type II bugs for which the prediction method suggests at least one candidate of supplementary change location. The denominator represents the total number of Type II bugs.

4. OBSERVATIONS

4.1 Observation 1: While a single trait is inadequate, combining multiple traits is limited as well.

To investigate whether repeated patterns of relationship exist between the initial and supplementary change locations, we identify frequently occurring paths between them, which can generalize the relationship. For bugs that are fixed more than once (Type II bugs), we parse the initial and supplementary patches to identify corresponding locations at the method level granularity to match them to corresponding method nodes in the CRG.

For the initial and supplementary change locations, we first investigate whether they can be connected with one hop of structural, historical, code clone, or name similarity relationships. The one hop relationships can represent existing change recommendation approaches. For example, two locations connected by a code clone edge means that the relationship between the two locations can be identified by code clone analysis.

We find that only 20%, 14%, and 10% of supplementary change locations can be reached within one hop from the initial change locations. This results indicate that remaining 80%, 86%, and 90% of supplementary change locations are not predictable using existing approaches in Eclipse JDT core, Eclipse SWT, and Equinox p2, respectively.

We also investigate the accuracy of prediction rules that are made using one relationship edge predicting supplementary change locations. We calculate the feedback, precision, and recall values of the rules made by one relationship edge, by applying them to the initial change locations of Type II bugs in order to predict corresponding supplementary change locations. Table 1 shows the results. Overall, the highest precision is only 8%, 9%, and 7% and the highest recall is only 17%, 22%, and 9%, in Eclipse JDT core, Eclipse SWT, and Equinox p2, respectively. These low accuracies indicate that a single trait is not adequate for predicting supplementary change locations.

Table 1: Feedback, precision, and recall of prediction rules made by one relationship edge

Relationship	Eclipse JDT core		
	feedback	precision	recall
calls	94.2%	1.9%	11.2%
called by	70.6%	7.7%	10.7%
code clone	29.2%	3.5%	1.1%
name similarity	84.8%	5.9%	16.4%
co-change	83.3%	4.2%	13.4%

Relationship	Eclipse SWT		
	feedback	precision	recall
calls	93.2%	0.8%	6.7%
called by	58.2%	7.0%	6.3%
code clone	51.3%	2.1%	1.7%
name similarity	82.8%	8.8%	11.2%
co-change	88.5%	6.0%	22.3%

Relationship	Equinox p2		
	feedback	precision	recall
calls	97.3%	1.6%	8.5%
called by	82.1%	6.9%	8.5%
name similarity	72.9%	2.9%	2.9%
co-change	68.4%	2.5%	6.9%

The relationship between the initial and supplementary change locations also can be represented using a combination of the CRG edges. For example, when an initial change location calls method X, and the method X has a code clone with a supplementary change location, the CRG can represent the relationship between them with a *calls* edge and a *code clone* edge. To investigate whether the combination of relationship edges can connect the initial and supplementary change locations, we study the number of supplementary change locations that are covered within n edges from corresponding initial change locations. Table 2 shows that our CRG can identify 30% to 33%, 24% to 33%, 16% to 19%, and 2% to 11% of supplementary change locations with two, three, four, and five relationship edges, respectively. Al-

Table 2: The portion of supplementary change locations that are covered within n edges from corresponding initial change locations.

# of edges	Eclipse JDT core	Eclipse SWT	Equinox p2
1	769 (20.0%)	1371 (13.5%)	240 (10.2%)
2	1270 (33.1%)	3170 (31.3%)	702 (29.7%)
3	906 (23.6%)	3353 (33.1%)	564 (23.9%)
4	626 (16.3%)	1961 (19.3%)	458 (19.4%)
5	196 (5.1%)	194 (1.9%)	260 (11.0%)
over 5	69 (1.8%)	94 (0.9%)	138 (5.8%)

though there are 2%, 1%, and 6% of supplementary change locations that cannot be connected within five edges from initial change locations, 98%, 99%, and 94% of them can be represented within five edges in Eclipse JDT core, Eclipse SWT, and Equinox p2, respectively.

We investigate the feedback, precision, recall, and f-score of the rules made by one to three relationship edges. F-score is a common measure assessing the predictive power which considers both precision and recall, and it is defined by $2*(precision*recall)/(precision+recall)$. We apply the rules to the initial change locations of Type II bugs to predict corresponding supplementary change locations. Overall, the f-score is at most 9%, 10%, and 8% with the precision value of 8%, 9%, and 5% and the recall value of 11%, 11%, and 20% in Eclipse JDT core, Eclipse SWT, and Equinox p2, respectively. These low f-scores indicate that combining of multiple traits does not predict supplementary change locations based on initial change locations any more accurately than single traits.

4.2 Observation 2: A boosting approach does not significantly improve the accuracy.

To improve the prediction accuracy, we hypothesize that combining the past accuracy information may improve the accuracy of future prediction. We divide the supplementary patch data set into a training set and an evaluation set as Table 3 shows. The training set is used to calculate the prediction accuracy of each rule. Based on the accuracy information of the training set, we use a boosting approach to predict supplementary change locations.

Table 3: The period of the training set and evaluation set

	Training period	Evaluation period
Eclipse JDT core	2002/01 ∼ 2006/08	2006/09 ∼ 2007/09
Eclipse SWT	2002/01 ∼ 2006/08	2006/09 ∼ 2007/09
Equinox p2	2006/10 ∼ 2008/10	2008/11 ∼ 2009/11

Boosting is a machine learning technique, that combines weak learners to create a strong learner [13]. To classify a new item, the boosting technique combines a set of results that are generated from weak learners by weighting them based on the accuracy of the weak learners in the training set. We develop a boosting approach that uses prediction rules as weak predictors. For a given initial change location, our boosting approach calculates a *prediction score* for each connected node within three edges of the initial change location. The prediction score is calculated by summing up the trained precision of the prediction rules corresponding to the paths to each node from the initial change location. The candidate locations are ranked with the prediction score; then, our boosting approach suggests *top N* nodes.

Table 4 shows the accuracy of the boosting approaches with different *top N* values. The results show that the precisions are 7%, 5%, and 6% and the recalls are 5%, 7%, and 10% in Eclipse JDT core, Eclipse SWT, and Equinox p2, respectively, even when we suggest only the three nodes that have the highest prediction score. We conclude that this boosting approach based on the past prediction accuracy also cannot accurately predict supplementary change locations.

Table 4: The accuracy of a boosting approach

top N	Eclipse JDT core		Eclipse SWT		Equinox p2	
value	prec.	recall	prec.	recall	prec.	recall
3	6.93%	4.77%	5.21%	7.35%	6.40%	9.93%
5	7.01%	8.65%	4.38%	9.89%	5.28%	12.80%
10	6.75%	14.58%	4.69%	16.20%	3.76%	16.29%
50	4.23%	21.05%	2.06%	29.47%	1.66%	29.18%
100	2.96%	29.01%	1.38%	38.20%	1.23%	33.70%
200	2.09%	51.23%	0.94%	49.48%	0.95%	40.21%

4.3 Observation 3: There is no package or developer specific pattern.

We hypothesize that filtering prediction approaches using package or developer specific information can improve the accuracy of the prediction. Package specific rules can improve the prediction accuracy when the packages have a repeated pattern of relationships between initial and supplementary change locations. Similarly, if a developer makes similar types of mistakes repeatedly, developer specific rules can improve the prediction accuracy.

We define package specific rules using a pre-condition and a relationship, as shown in the example below:

Pre-condition: *if the initial change location is in* org-.eclipse.jdt.core.util
Relationship: *suggest locations that have been co-changed with the initial change location.*

Developer specific rules are defined in the similar way; the pre-condition forms *if the initial change is committed by the developer* Tom.

We firstly identify every package and developer specific rule that appears at least once in the training set. To make package and developer specific prediction rules, we gather the pre-conditions of the initial changes (package names and committer names) and identify the relationship between the initial and supplementary change locations of Type II bugs in the training set. We then calculate the accuracy of the prediction rules in the training set.

Figure 1 plots the feedback and precision of the three kinds of prediction rules in Eclipse JDT core. We find that general rules show high feedback but low precision, and developer/package specific rules show low feedback but high precision in all of the study subjects. This is quite natural, because when we filter the information according to a specific package or developer, the rules can precisely predict supplementary change locations for fewer applicable initial change locations.

Based on the trained accuracies, we develop boosting approaches similar to those in the previous section. We find that the boosting approaches based on package and developer specific rules do not improve the prediction accuracy; indeed, their accuracy is even lower than that of the boosting approach based on general rules in some cases. Overall, the highest improvements of the accuracy (in terms of f-score) compared to the boosting approach based on general rules are 1.24%, 1.02%, and 0.78% for the boosting approach based on package specific rules, and 0.92%, 0.44%, and 1.20% for the boosting approach based on developer spe-

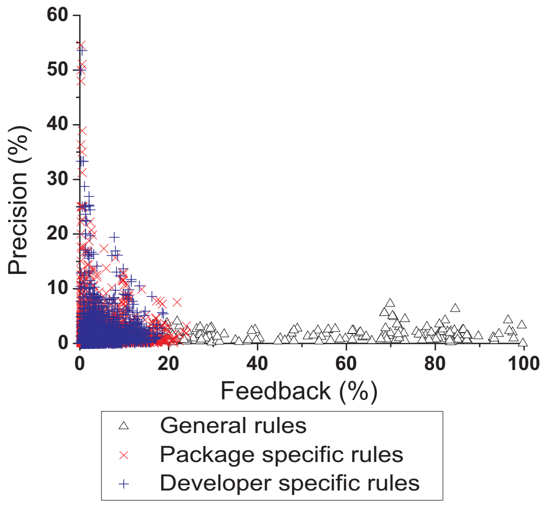

Figure 1: Feedback vs. precision on three kinds of prediction rules (in Eclipse JDT core)

cific rules in Eclipse JDT core, Eclipse SWT, and Equinox p2, respectively.

Our sub-conclusion here is that no developer or package specific pattern between initial and supplementary change locations exists.

4.4 Observation 4: There is no repeated mistake.

Although we cannot find a repeated pattern of relationship between initial and supplementary change locations based on existing structural dependency, historical co-change, code clone, and name similarity relationship, there might be an uncovered relationship which can result in repeated patterns. In this section, we investigate the patterns between initial and supplementary change locations.

In a similar association rule mining approach of previous work [3, 16, 19], but applying them to the supplementary patch data set, we can represent the pattern between initial and supplementary change locations as a rule:—*"If* methodA *is changed in an initial change, then* testMethodA *is changed in the supplementary change."*

We investigate whether there is a repeated pattern of the initial and supplementary change location pairs. If the initial change locations are {A, B} and the supplementary change locations are {X, Y}, we can make the following pattern rules—$(A \rightarrow X), (A \rightarrow Y), (B \rightarrow X), (B \rightarrow Y)$. We identify the number of occurrence for each pattern.

The results shown in Table 5 indicate that the majority of patterns (78%, 96%, and 95% in Eclipse JDT core, Eclipse SWT, and Equinox p2, respectively) appear only once, and under 2% of patterns appear more than three times in the study subjects. This result indicates that we cannot generate an appropriate suggestion based on the majority of the patterns, because they have occurred only once.

In addition, we also find that the same location does not require supplementary fixes repeatedly. Table 6 shows the portion of initial change locations appearing n times. 69%, 71%, and 84% of initial change locations appear only once in the version history in Eclipse JDT core, Eclipse SWT, and Equinox p2, respectively. These results indicate that developers rarely make repeated mistakes at the same location; version history based pattern mining cannot be accurate at finding supplementary change locations.

Table 5: The number of patterns between initial and supplementary change locations appearing n times

Number	Eclipse JDT core	Eclipse SWT	Equinox p2
1	73455 (77.5%)	116601 (96.0%)	32715 (94.9%)
2	19412 (20.5%)	4383 (3.6%)	1765 (5.1%)
3	755 (0.8%)	312 (0.3%)	11 (0.0%)
4	948 (1.0%)	45 (0.0%)	0 (0.0%)
over 4	207 (0.2%)	127 (0.1%)	0 (0.0%)

Table 6: The number of initial change locations appearing n times

Number	Eclipse JDT core	Eclipse SWT	Equinox p2
1	2704 (68.8%)	2877 (71.3%)	1988 (84.2%)
2	810 (20.6%)	680 (16.9%)	302 (12.8%)
3	235 (6.0%)	243 (6.0%)	45 (1.9%)
4	77 (2.0%)	114 (2.8%)	23 (1.0%)
over 4	106 (2.7%)	119 (3.0%)	3 (0.1%)

5. DISCUSSION

We consider how the supplementary change locations can be identified when the initial change is given. The accuracy predicting the supplementary change location is already low, but the problem can be more difficult in practice, because we need to identify the changes that require supplementary patches. We can identify incomplete patches based on machine learning techniques (e.g., SVM), by investigating the characteristics of the incomplete patches (e.g., date, committer, contents of the patch, etc.).

We do not compare prediction rules to see which one is superior to the others, because the overall accuracies of the rules are low. We find that *called by* relationship is more accurate than the other relationships (see Table 1). This result implies that *called by* relationship is harder for programmers to detect than the other relationships. For a given initial change location, developers should trace the call hierarchy to find caller method of a method in an initial change location. Because developers are more likely to miss the entities connected by *called by* relationship to the initial change, the relationship occurs more frequently between initial and supplementary changes than other relationships.

Regarding threats to validity, different experimental settings can affect our prediction accuracy. For example, instead of using all history data as a training set of the boosting approach, we can use only recent information as a training set to limit the effect of the old data. In addition, fuzzy logic, random forests, or neural network can be used instead of a boosting approach. Furthermore, we use only sub-projects of the Eclipse project as our study subjects which are written mostly in Java. Different experimental settings might improve prediction accuracy, but we doubt that there is a silver bullet that can resolve this problem of reducing omission errors.

6. CONCLUSION

Since about ten years ago, when guiding software changes based on mining version histories showed early promise, many change recommendation systems have been proposed to identify additional change locations given an existing change set. Our study is the first systematic and comprehensive investigation of a real-world supplementary patch data set, where developers missed updating the entities together with the initial change. In this paper, using the supplementary change data set, we develop a novel representation, called *change relationship graph (CRG)*, that allows us to investigate the relationship between two program locations based on a combination of structural, historical, name similarity, and code clone relationships. Based on the CRG, we investigate why it is inherently challenging to predict supplementary change locations given initial change locations.

Through a comprehensive study, we observe that while no single rule is adequate, combining multiple rules is limited as well. A boosting approach using the rules does not show a high accuracy; rather, it shows that past accuracy information on a training set does not improve future prediction accuracy in the evaluation set. Beyond this, neither developer nor package specific information is found to improve the accuracy. Moreover, there is no repeated pattern between initial and supplementary change locations, and developers do not make omission errors at the same locations repeatedly. As researchers who participated in the community of mining software repository, we share our skepticism that reducing real-world omission errors based on a systematic evaluation of a supplementary patch data set is inherently challenging.

7. ACKNOWLEDGEMENT

This work was partly supported by the IT R&D program of MSIP/KEIT [10041313, UX-oriented Mobile SW Platform]. This work was supported in part by the National Science Foundation under grants CCF-1149391, CCF-1117902, SHF-0910818, CCF-1018271, CCF-0811524, CNS-1239498, and a Google Faculty Award.

8. REFERENCES

[1] B. Dagenais and L. Hendren. Enabling static analysis for partial java programs. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 313–328, New York, NY, USA, 2008. ACM.

[2] A. E. Hassan and R. C. Holt. Predicting change propagation in software systems. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 284–293, Washington, DC, USA, 2004. IEEE Computer Society.

[3] K. Herzig and A. Zeller. Mining cause-effect-chains from version histories. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pages 60–69. IEEE, 2011.

[4] H. Kagdi, S. Yusuf, and J. I. Maletic. Mining sequences of changed-files from version histories. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 47–53, New York, NY, USA, 2006. ACM.

[5] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on*, 28(7):654–670, 2002.

[6] H. Malik and A. E. Hassan. Supporting software evolution using adaptive change propagation heuristics. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 177–186. IEEE, 2008.

[7] S. Mirarab, A. Hassouna, and L. Tahvildari. Using bayesian belief networks to predict change propagation in software systems. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*, pages 177–188. IEEE, 2007.

[8] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 315–324, New York, NY, USA, 2010. ACM.

[9] J. Park, M. Kim, B. Ray, and D.-H. Bae. An empirical study of supplementary bug fixes. In *MSR '12: 9th IEEE Working Conference on Mining Software Repositories*, pages 40 –49, Washington, DC, USA, june 2012. IEEE Computer Society,.

[10] M. K. Ripon Saha, Ray Qiu and D. Perry. A graph-based framework for reasoning about relationships among software modifications. Technical report, 2014.

[11] M. P. Robillard. Automatic generation of suggestions for program investigation. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 11–20, New York, NY, USA, 2005. ACM.

[12] Z. M. Saul, V. Filkov, P. Devanbu, and C. Bird. Recommending random walks. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 15–24. ACM, 2007.

[13] R. E. Schapire. The strength of weak learnability. *Machine learning*, 5(2):197–227, 1990.

[14] M. Song and M. Kim. A query-by-example approach for searching related software revisions. Technical report, 2014.

[15] Z. Xing and E. Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 54–65, New York, NY, USA, 2005. ACM.

[16] A. T. T. Ying, G. C. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

[17] Y. Zhou, M. Wursch, E. Giger, H. Gall, and J. Lu. A bayesian network based approach for change coupling prediction. In *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on*, pages 27–36. IEEE, 2008.

[18] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.

[19] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.