Towards Efficient and Effective Automatic Program Repair

Xuan-Bach D. Le
School of Information Systems
Singapore Management University, Singapore
dxb.le.2013@phdis.smu.edu.sg

ABSTRACT

Automatic Program Repair (APR) has recently been an emerging research area, addressing an important challenge in software engineering. APR techniques, if *effective* and *efficient* , can greatly help software debugging and maintenance. Recently proposed APR techniques can be generally classified into two families, namely search-based and semantics-based APR methods. To produce repairs, search-based APR techniques generate huge populations of possible repairs, i.e., *search space*, and lazily search for the best one among the search space. Semantics-based APR techniques utilize constraint solving and program synthesis to make search space more tractable, and find those repairs that conform to semantics constraints extracted via symbolic execution. Despite recent advances in APR, search-based APR still suffers from search space explosion problem, while the semantics-based APR could be hindered by limited capability of constraint solving and program synthesis. Furthermore, both APR families may be subject to *overfitting*, in which generated repairs do not generalize to other test sets.

This thesis works towards enhancing both effectiveness and efficiency in order for APR to be practically adopted in foreseeable future. To achieve this goal, other than using test cases as the primary criteria for traversing the search space, we designed a new feature used for a new search-based APR technique to effectively traverse the search space, wherein bug fix history is used to evaluate the quality of repair candidates. We also developed a deductive-reasoning-based repair technique that combines search-based and semantics-based approaches to enhance the repair capability, while ensuring the soundness of generated repairs. We also leveraged machine-learning techniques to build a predictive model that predicts whether an APR technique is effective in fixing particular bugs. In the future, we plan to synergize many existing APR techniques, improve our predictive model, and adopt the advances of other fields such as test case generation and program synthesis for APR.

CCS Concepts

•**Software and its engineering** → Software creation and management;

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2975934

Keywords

Automatic Program Repair, Deductive Reasoning, Mining Software Repository, Genetic Programming

1. INTRODUCTION

Software bugs are one of the primary challenges in software development, which usually incur significant cost in software production. Given short time to market, mature commercial software systems are delivered with both known and unknown bugs [15], despite the support of multiple developers and testers dedicated for such projects. To maintain the quality of software, bug fixing is thus an important task. Yet, it is notoriously hard, time-consuming, and laborious process, which usually dominates developer time [29], and software maintenance cost [7]. Thus, there is a dire need to automate bug fixing process to help reduce the burden and cost on software debugging and maintenance.

Substantial recent works on Automatic Program Repair (APR) have been proposed to repair real-world software, making the once-futuristic idea of APR become gradually materialized. These repair techniques generally fall into two categories: search-based methodology (e.g., GenProg [6], PAR [9], SPR [16], Prophet [18], and HDRepair [14]) and semantics-based methodology (e.g., SemFix [22], Nopol [4], DirectFix [19], and Angelix [20]). Search-based repair techniques generate a large pool of possible repair candidates, i.e., search space of repairs, and then search for correct repair within that search space using an optimization function. Meanwhile, semantics-based techniques leverage constraint solving and program synthesis to generate repairs that satisfy semantics constraints extracted via symbolic execution and provided test suites.

Despite recent advances in APR, current approaches are limited in several ways [17, 23]. To truly make APR adoptable in practice, an APR technique must be *effective* (i.e., able to correctly fix many bugs), and *efficient* (i.e., able to do so at affordable costs). Although being able to fix general bugs potentially renders search-based APR more effective, the efficiency of these techniques could be hindered by the search space explosion problem, wherein correct repair candidates sparsely occur [17], which may consume several hours to complete the search for correct repairs [6]. Semantics-based APR, on the other hand, attempts to fix less generic bugs on some restricted categories, e.g., assignments, if-conditions, and makes use of program synthesis with restricted components (like in [1]) to render the search space more tractable. Thus, it is more efficient than search-based APR (e.g., Angelix can fix 28 bugs in 32 minutes on average). However, its effectiveness could be hampered by limited capability of constraint solving and program synthesis (e.g., non-linear constraints are hard to solve). Furthermore, both search-based and semantics-based APR may generate "plausible" repairs

– those that do not generalize beyond the provided test suites, due to the incomprehensiveness of the given test suites.

The goal of this research is to work towards enhancing both effectiveness and efficiency in order for APR to be potentially adopted in practice. To achieve this goal, we have done the followings: (1) First, we developed a *history driven repair* technique, namely HDRepair [14], that performs far better than its counterparts in the same search-based APR family. The results also showed that HDRepair generates good-quality repairs at reasonable time cost. (2) Second, we developed a repair technique based on deductive reasoning to marry the strengths of both search-based and semantics-based APR together [11]. While search-based APR alone may be ineffective and inefficient due to a sparse search space [17], cultivating the search space with repair ingredients generated by light-weight semantics analysis could help condense correct repairs in the search space. (3) Third, to better utilize the wealth of APR techniques, we developed a predictive model that is able to predict whether an APR technique is effective in fixing particular bugs [12], helping developers choose the best APR that suites their need.

The structure of the remainder of this paper is as follows. Section 2 describes contributions that the thesis contributes to the field. Section 3 describes discussion and future work of the thesis, including synergizing many existing APR techniques, bug fixes' specification mining, improving predictive model for APR's effectiveness, and tests generation and program synthesis. Section 4 concludes the paper.

2. CONTRIBUTIONS

This section provides further details on (published) research work on program repair that the thesis focuses on.

2.1 History Driven Program Repair

A recent study on program repair showed that test suites alone may not be adequate for APR techniques to generate correct repairs [17], addressing the important need of seeking other features that could help APR generalize.

To this end, we proposed to use bug fix history to effectively guide and drive the program repair process [14]. An important feature that differentiates our new technique from previous work is that it evaluates the fitness or suitability of a repair candidate by the degree to which it is similar to various prior bug-fixing patches in the bug fix history. Also, different from work like PAR which mines bug fix history to *construct* and apply repair templates, we use fix history as the primary criteria for assessing potential *quality* of generated repair candidates. This is in contrast with previous APR techniques, which by and large use only input test suites as the only criteria for assessing patch suitability. [1] The main intuition for using bug fix history to inform patch suitability/quality is that bug fixes are often similar in nature and past fixes can be a good guide for future fixes [2, 9].

```
//Human fix: fa * fb > 0.0
if (fa * fb >= 0.0 ) {
   throw new ConvergenceException("...")
}
```

Figure 1: Bug in Math version 85

To illustrate, consider the buggy code snippet in Figure 1, taken from Math version 85 in Defects4J benchmark [8]. The *ConvergenceException* will be thrown when one of the test cases is executed, causing the test to fail. One naive way to "fix" the problem is to simply delete the *throw* statement, which eliminates the bad symptom and causes the failing test to trivially pass. However, this is a nonsensical solution. Unfortunately, prior APR techniques would not be able to identify such a solution as nonsensical due to the reliance on test cases as the only yardstick for patch quality. Our history driven approach, on the other hand, can effectively identify those nonsensical patches since those candidate patches rarely occur in the history, and thus would be received very low patch-suitability scores. Those candidates that receive low scores are likely to be filtered out during the search for repair in our approach.

[1]We use the words "patch" and "repair candidate" interchangeably

Our history driven APR technique works on there phases: (1) *bug fix history extraction* (2) *bug fix history mining* and (3) *bug fix generation*. In the first phase, historical bug fixes are mined from several revision control systems of hundreds of projects in GitHub. In the second phase, we mine the frequent bug fix patterns in the historical data extracted in the first phase. These mined bug fix patterns are then used as knowledge base for the third phase. In the third phase, we iteratively generate repair candidates using existing mutation operators borrowed from program repair and mutation testing. Candidates' suitabilities are then assigned by querying the constructed knowledge base, e.g., candidates that match many frequent bug fix patterns in the knowledge base would receive higher scores. Candidates with higher suitability scores are more likely to be chosen, and ranked higher during the search process. Finally, we return a ranked list of possible repair candidates that pass previously failed test cases as recommendation to developers.

We performed an experiment on HDRepair, PAR, and GenProg on 90 bugs taken from Defects4J dataset. As compared to the baselines, the advantage of HDRepair is two-fold: better effectiveness and efficiency. That is, HDRepair repairs far more bugs (23 bugs) than PAR (4 bugs) and GenProg (1 bug) can repair. Also, HDRepair takes 20 minutes on avarage to generate fixes for these bugs. Manual inspection of the results suggests various reasons for why HDRepair performs better than others. This includes the power of mutation operators, time needed to generate repairs, and the correctness of repairs. That is, GenProg and PAR sometimes lack the necessary mutation operators to create repairs, causing the techniques to timeout without finding any repairs. Other than that, GenProg and PAR generate incorrect repairs, which trivially cause all provided tests to pass but fail to match with the desired behaviours as patches submitted by developers. HDRepair, on the other hand, utilizes the strengths of many mutation operators to diversify the set of possible candidates, and leverages bug fix history to assess suitability of generated candidates. Thus, HDRepair normally is able to find repairs that are close to the developers' fixes. In the future, we expect that the history-driven approach, in complement with test-driven approaches, would be able to help avoid generating "plausible" fixes (also known as overfiting).

2.2 Program Repair Based on Deductive Reasoning

Test-driven and history-driven repair approaches generate patches that are susceptible to overfitting. In other words, generated patches may not be correct in the sense that they are not automatically verified for correctness, but rather relying on developers/experts for correctness assessment.

We address the problem of patch correctness/soundness by utilizing a deductive reasoning method [11]. Given a buggy program with specifications written in Separation Logic [24], we generate *syntactic* and *semantics* repair candidates. Syntactic candidates are generated by borrowing mutation operators from GenProg to syntactically change the original program, e.g., delete a statement, replace a statement with another, etc. Semantic candidates are generated from the provided specifications as repair templates, e.g., expected output values, conditional structure (*if-then-else*), etc. The reason for incorporating syntactic and semantic candidates together is that these candidates could help one another in tandem to condense the search space with more valuable candidates. For example, the search space with syntactic repair candidates alone is sometimes very sparse [17], since real fixes could lie beyond the syntactic candidates. Semantic candidates, however, could cultivate the search space with more useful candidates. With this combined search space, we then validate candidates by an underlying deductive verifier [3], and select best candidates, which receive fewer number of warnings issued by the verifier, for patch evolution through genetic programming. The process is repeated until a patch that receives no warnings from the verifier is found. This way, patches generated by our system are automatically guaranteed correct, following the correctness of the underlying verifier.

Table 1 shows the details of the 10 bugs in our experiments with our approach and Angelix – a current state-of-the-art semantics-based APR. In the table, "LOC" column shows the number of lines in each program (including specification lines). The "Mutated Loc" column indicates the name of the function where the bug is seeded. In this column, "dupp" stands for do upgrade process prio, "ncl" stands for numeric case loop, and "IBC" is a short form of Inhibit Biased Climb . The "Time" column shows the time (in minutes) needed by our approach to fix the bugs. The "Category" column shows the type of the error, either missing implementation or incorrect implementation. We note that we reused SIR program specifications constructed by Le *et al.* [10] and injected seeded bugs to the original programs by mutating the original programs in various different ways.

Initial experiment results show that our approach can fix these 10 bugs, while Angelix can only fix one out of these bugs (tcas2). For all 10 bugs except one, our approach can successfully fix the bugs in less than 3 minutes. The bug that took the most time to fix was the print tokens bug which requires 6.25 minutes. Our further observation on the results is that Angelix by default cannot fix bugs that require adding new statements. That is, its synthesis engine is geared towards synthesizing only guards, assignments, if-conditions, and loop-conditions, and is not able to produce new code out of thin air, e.g., inserting new statements. Also, some bugs involve nonlinear constraints, which are hard for Angelix's synthesis engine to solve. Our cultivated search space including both syntactic and semantic candidates, however, comes in handy in this place. That is, our approach can generate candidates that lie beyond the search space of Angelix, and thus is more likely to be able to find the correct fixes with the help of the optimization function which assesses patch suitability by the number of warnings issued by the underlying verifier. In the future, we expect a greater success of combining syntactic and semantic repair candidates, to potentially enhance repair capability of APR and ensure patch quality.

2.3 Predicting Effectiveness of APR

Recent years have seen the proliferation of many different APR techniques with various strengths and weaknesses, e.g., one might fix bugs that others cannot ever fix. To better utilize the wealth of APR, we developed a predictive model that is able to predict whether an APR technique is effective at fixing particular bugs [12]. Our idea is somewhat similar to the idea of predicting correct program in programming by example proposed by Gulwani *et al.* [27], whereby correct programs that meet user-defined specifications (e.g., input-output examples) are ranked higher in the search space of possible candidates. In our setting, we would like to suggest the best APR technique that could fix given bugs from users. Thus, users can either proceed with the effective APR returned by our predictive model, or switch to traditional way of manually fixing bugs rather than desperately hoping for an inefficient APR technique to automatically fix the bugs for them.

Table 1: **Evaluation of our proposed approach on programs from the SIR benchmark with manually seeded bugs.**

Program	Mutated Loc	LOC	Time	Category
uniq	gline_loop	74	0.5	Incorrect
replace	addstr	855	2.8	Missing
replace	stclose	855	2.15	Missing
replace	stclose	855	2.2	Incorrect
replace	locate	855	2.5	Incorrect
replace	patsize	855	0.5	Incorrect
replace	esc	855	2.14	Incorrect
schedule3	dupp	693	0.43	Incorrect
print_tokens	ncl	1002	6.25	Missing
tcas2	IBC	302	0.15	Incorrect

Our predictive model was built for search-based APR approaches, e.g., GenProg. At an early stage of running the repair tool, we extract a number of features that are potentially related to the effectiveness of the tool, e.g., the size of input program, the number of test cases, etc. We then leverage machine learning technique to process these features and learn a discriminative model that can predict whether continuing the search process would result in a repair within a desired time limit. It is worth noting that the features are extracted only a few seconds after the repair tool is run. We perform our experiment on 105 real-world bugs taken from [6], in which GenProg can generate fixes for 53 bugs. Experiment results showed that our model can identify effective cases from ineffective ones (i.e., bugs for which the repair tool cannot produce correct fixes after a long period of time) with a precision, recall, F-measure, and AUC of 72%, 74%, 73%, and 76%, respectively. Comparing with other studies solving various prediction tasks in the software engineering research literature, e.g., [25, 26, 21], the performance of our approach is comparable or higher. We also highlighted the important features that help our model achieve good performance. These include the diversity of the pool containing possible repair candidates generated by the repair tool, the number of faulty areas considered to be fixed, the input program and test cases, etc.

3. DISCUSSION & FUTURE WORK

Currently, the test-driven and history-driven techniques are being developed alone, leaving the potentially interesting combination of them untapped. We envision, however, that test-driven and history-driven approaches can significantly help one another to mitigate current issues with APR. Particularly, our history-driven technique could potentially help test-driven approaches to mitigate the overfitting issue, since history-based approach does not directly use test cases as the primary criteria for assessing patch suitability. Test-driven approaches, e.g., Angelix, on the other hand, can help cultivate the search space with more useful repair candidates. Thus, synergizing these techniques could be an interesting line of work that this thesis would tackle in the future.

Program repair based on deductive reasoning, wherein search-based and semantics-based APR are merged together, has shown promising results. However, complete specifications are required in order to completely ensure the soundness of generated patches.

We expect that employing advances in specification inference and mining could help reduce this burden. In future work, we would like to use the history of bug fixes to mine specifications of past patches and inform the future patch generation process in a way that generated patches are likely to conform to desired behaviours.

Our predictive model is currently designed specifically for search-based APR techniques. Future work would be to design more useful features, e.g., features that characterize patch quality, to generalize our model for both search-based and semantics-based APR techniques. Also, we could augment our model to further help APR predict correct patches in a huge search space of possible candidates as similarly proposed by program synthesis work in [27]. This potentially helps render the search space of APR more tractable since correct patches could be more easily identified among the search space, increasing both efficiency and effectiveness of APR techniques.

Automated test case generation and program synthesis have been proven helpful in the domain of program repair [28, 20, 13]. We believe that further tapping into these research areas would be an interesting direction to help advance current APR techniques. For instance, tests generation tools could help generate more comprehensive test suites, and thus would help mitigate the overfitting issue, e.g., a well-tested patch would less likely to be a "plausible" one. Incomprehensive test suites may render program synthesis generate spurious solutions. Strong program synthesis techniques, e.g., lazy synthesis [5], would help semantics-based APR techniques, e.g., Angelix, cope with incomplete specifications, making APR amenable to incomprehensive test suites. An interesting future work would be to explore the effectiveness of many different program synthesis techniques, and find the best way to adopt the synthesis techniques for APR.

4. CONCLUSION

Automated program repair addresses an important challenge in software engineering. Recent years have seen this field growing rapidly with many proposed techniques. Our research contributes to the field in several aspects to mitigate current issues with APR, e.g., efficiency and effectiveness. In sum, we developed a new history-based APR technique that performs far better than other techniques in the same family. We also developed a deductive-reasoning-based APR technique, which merges search-based and semantics-based approach together, for ensuring patch correctness. To better utilize many existing APR techniques, we developed a predictive model that is capable of predicting APR's effectiveness, assisting users in choosing the best APR that suites their need, e.g., fixing bugs in a desired time budget. As future work, we will continue working towards enhancing both effectiveness and efficiency of APR, rendering it to be adopted in practice in foreseeable future.

5. REFERENCES

[1] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. 2015.

[2] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In *FSE*, 2014.

[3] W.-N. Chin, C. David, and C. Gherghina. A hip and sleek verification system. In *OOPSLA companion*, 2011.

[4] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *CSTVA*, 2014.

[5] B. Finkbeiner and S. Jacobs. Lazy synthesis. In *VMCAI*, 2012.

[6] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *TSE*, 2012.

[7] P. Hooimeijer and W. Weimer. Modeling bug report quality. In *ASE*, 2007.

[8] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *ISSTA*, 2014.

[9] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *ICSE*, 2013.

[10] Q. L. Le, A. Sharma, F. Craciun, and W.-N. Chin. Towards complete specifications with an error calculus. In *NFM*, 2013.

[11] X. B. D. Le, Q. L. Le, D. Lo, and C. L. Goues. Enhancing automated program repair with deductive verification. In *ICSME*, 2016.

[12] X.-B. D. Le, T.-D. B. Le, and D. Lo. Should fixing these failures be delegated to automated program repair? In *ISSRE*, 2015.

[13] X. B. D. Le, D. Lo, and C. L. Goues. Empirical study on synthesis engines for semantics-based program repair. In *ICSME*, 2016.

[14] X. B. D. Le, D. Lo, and C. Le Goues. History driven program repair. In *SANER*, 2016.

[15] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. *ACM SIGPLAN Notices*, 2003.

[16] F. Long and M. Rinard. Staged program repair in spr. 2015.

[17] F. Long and M. Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *ICSE*, 2016.

[18] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *POPL*. ACM, 2016.

[19] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *ICSE*, 2015.

[20] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *ICSE*, 2016.

[21] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *CCS*, 2007.

[22] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: program repair via semantic analysis. In *ICSE*, 2013.

[23] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA*, 2015.

[24] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, 2002.

[25] H. Seo and S. Kim. Predicting recurring crash stacks. In *ASE*, 2012.

[26] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto. Studying re-opened bugs in open source software. *EMSE*, 2013.

[27] R. Singh and S. Gulwani. Predicting a correct program in programming by example. In *CAV*, 2015.

[28] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *FSE*, 2015.

[29] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *MSR*.