

GLAD: Neural Predicate Synthesis to Repair Omission Faults

Sungmin Kang
KAIST
sungmin.kang@kaist.ac.kr

Shin Yoo
KAIST
shin.yoo@kaist.ac.kr

Abstract—Existing template and learning-based Automated Program Repair (APR) tools have successfully found patches for many benchmark faults. However, our analysis of existing results shows that omission faults pose a significant challenge. For template based approaches, omission faults provide no location to apply templates to; for learning based approaches that formulate repair as Neural Machine Translation (NMT), omission faults similarly do not provide faulty code to translate. To address these issues, we propose GLAD, a novel learning-based repair technique that targets if-clause synthesis. GLAD does not require a concrete faulty line as it is based on generative Language Models (LMs) instead of machine translation; consequently, it can repair omission faults. To provide the LM with project-specific information critical to synthesis, we incorporate two components: a type-based grammar that constrains the model, and a dynamic ranking system that evaluates candidate patches using a debugger. Our evaluation shows GLAD is highly orthogonal to existing techniques, correctly fixing 26 Defects4J v1.2 faults that previous NMT-based techniques could not, while maintaining a small runtime cost, underscoring its potential as a lightweight tool to complement existing tools in practice. An inspection of the bugs that GLAD fixes reveals that GLAD can quickly generate expressions that would be challenging for other techniques.

Index Terms—Program Repair, Machine Learning, Debugging

I. INTRODUCTION

In Automated Program Repair (APR), one seeks to automatically fix faulty code given a specification of desired behavior. As APR is widely researched, it would be beneficial to look at the performance of the community as a whole. As many APR techniques are evaluated on the Defects4J [1] dataset, we can assess the collective performance of APR tools by analyzing which faults have been fixed by any tool that deals with Defects4J at the time of this writing. More importantly, our analysis reveals that the amount of *added characters* in the developer patch is strongly indicative of APR success: while 93.8% of faults that require fewer than 16 characters to be added are fixed, only 23.7% of faults that require more than 32 characters to be added were ever fixed.

A large portion of the patches that add a large number of characters contain *omission faults*, i.e. faults in which necessary code is missing. Omission faults pose a problem to both template-based and learning-based techniques. Template-based APR tools may fix only a restricted set of omission faults such as adding null checks, as there is no faulty code to apply templates to. Meanwhile, learning-based techniques use the neural machine *translation* formulation, which necessitates

a faulty statement to translate to a fixed statement, making omission faults difficult to fix. On the other hand, language models (LMs) are a natural solution to repairing omission faults, as they are trained with a generative loss [2] and are consequently better suited to handle omission faults. To this end, we introduce GLAD (Grammar-based LAnguage models with Debuggers) which specializes in fixing `if` statement omission faults.

We perform empirical analysis to demonstrate the utility of GLAD. In RQ1, which investigates which bugs GLAD repairs, we find that GLAD could fix 26 faults in Defects4J v1.2 that previous learning-based techniques could not, and overall we report fixing eleven faults that were never fixed by any APR tool we examine, and that GLAD is capable of fixing bugs in Defects4J v2.0, implying the versatility of GLAD's repair performance. In RQ2, which investigates the influence of each component, our results show both the grammar and the debugger-based reranking make a significant contribution towards reducing the space of patches to evaluate, showing that incorporating project-specific information is helpful. Qualitatively inspecting the bugs that GLAD uniquely fixes, we find that they require the synthesis of expressions that are difficult to generate using existing techniques.

II. APPROACH

GLAD is a learning-based technique that specifically aims to repair `if`-statement omission faults. By moving away from the translation paradigm (i.e., learning how to translate a faulty statement to a correct statement), and instead using the context of the faulty location to directly synthesize code, GLAD is well-adapted to fixing omission faults.

There are six steps in the repair pipeline. Before repair proceeds, a pretrained language model, specifically a GRU recurrent neural network trained on data from 1,000 open-source Java repositories, is fine-tuned on the faulty version of the code so that the LM can pick up token usage patterns from the target project. Next, given a suspicious location, the body of the method preceding the faulty location is tokenized, and a 'repair seed', namely an `if` token along with an opening parenthesis, is added to the end. Prior to beam search, static information such as available identifiers and available members to types is extracted via a debugger, to construct a grammar constraining the language model. The fine-tuned language model takes cue from the injected `if` token and

finds predicate candidates using grammar-aware beam search to generate grammatically valid predicates. Generated predicates are evaluated by a debugger; the obtained dynamic information is used to filter and rank generated patches, based on how correct code should behave: the core intuition is that the generated patches must change failing test behavior, while preserving passing test behavior is preferable but not necessary. Finally, we add bodies to the predicates (using a mix of templates and further LM synthesis) and evaluate the patches over tests for verification. By incorporating generative models with project-specific information, we aim to repair bugs that prior techniques could not, as we evaluate in the next section.

III. RESULTS

A. Repair Performance (RQ1)

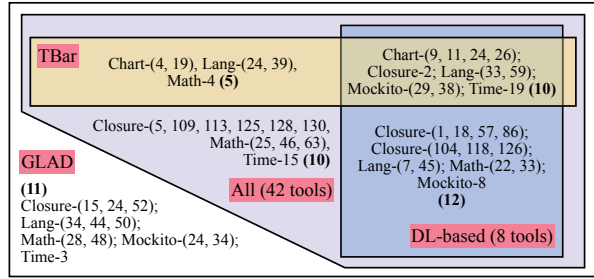


Fig. 1. Overlap analysis of bugs in the Defects4J benchmark correctly fixed by GLAD under perfect fault localization. The number of bugs in each category are in parentheses.

In our first research question, we evaluate the repair performance of GLAD, and compare it against baselines. Repair results on faults from Defects4J v1.2 are presented in Figure 1. GLAD can fix 48 faults correctly using perfect FL, and make plausible patches for an additional 53 bugs. More to the point is how *different* the bugs that GLAD fixes are relative to other techniques. Relative to TBar, GLAD fixes **33** different bugs correctly; comparing against bugs known to be fixed by deep learning-based techniques (DL-based), GLAD fixes **26** different bugs under the perfect fault localization scenario, demonstrating GLAD can fix a unique profile of bugs. Even when compared with all 42 APR tools we are aware of, overall GLAD fixes **11** bugs (GLAD) for the first time, underscoring the uniqueness of GLAD-generated solutions. All bugs that GLAD fixes for the first time contain synthesis elements such as strings (Lang-50) or long sequences of code (Lang-44) that were difficult for prior work to handle, showing that GLAD makes progress in handling omission faults, due to its synthesis capabilities. Meanwhile, GLAD could correctly repair 35 bugs in the Defects4J v2.0 benchmark, showing that its performance is not limited to the projects in the Defects4J v1.2 benchmark.

An example of a uniquely fixed bug is shown in Table I, where the generated fix for the Mockito-24 bug from the Defects4J benchmark is presented; GLAD successfully generates the correct patch, which requires the synthesis of a

TABLE I
CORRECT PATCH FOR BUG ONLY FIXED BY GLAD

Patch ID	Patch Content
Mo-24-dev	Add if(invocation.getMock() == invocation.getArguments()[0]) {return 0;}
Mo-24-GLAD	Add if(invocation.getMock() == invocation.getArguments()[0]) {return 0;}

significantly complex expression comparing the results of two distinct method calls.

B. Component Ablation Study (RQ2)

We performed an ablation study to confirm that each component indeed contributes to performance. First, without finetuning, the model generates a slightly smaller number of plausible patches, but the total number of faults correctly fixed drops by half, indicating that finetuning indeed helps the language model pick up within-project lexical patterns. Without grammar, only a few bugs are correctly fixed, while the number of plausible patches drops by about a third, as the LM is unaware of what expressions are acceptable within the given project. This suggests that LMs, and perhaps learning-based techniques in general, would greatly benefit from project-specific static information that can easily be acquired. Overall, these results show that the synergy between the LM and the grammar enables GLAD to generate complex expressions that fix previously unfixable faults. Meanwhile, our dynamic reranking enhances the efficiency of GLAD: based on the average time it took to verify a single patch, we estimate that on average 5 hours and 21 minutes of patch verification time to find the correct patch were saved, allowing GLAD to achieve an average repair time of 22 minutes.

IV. CONCLUSION

In this work, we propose GLAD, an APR technique that aims to fix omission faults, which are challenging for existing techniques. GLAD is highly orthogonal to existing work, fixing 26 faults never fixed by previous learning-based tools, and fixing eleven faults no other tool has fixed, often in under 30 minutes. Further, each component of GLAD contributes significantly to its performance, providing support for its design. We thus verify the utility of GLAD as a repair tool that is effective at fixing bugs within its domain; the orthogonality of GLAD makes it a good candidate for an ensemble of APR tools to deploy as well.

REFERENCES

- [1] R. Just, D. Jalali, and M. D. Ernst, “Defects4j: A database of existing faults to enable controlled testing studies for java programs,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 437–440. [Online]. Available: <https://doi.org/10.1145/2610384.2628055>
- [2] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, “Improving language understanding by generative pre-training,” 2018.