F1X at APR-COMP 2024

Sergey Mechtaev
University College London
United Kingdom
s.mechtaev@ucl.ac.uk

Shin Hwei Tan
Concordia University
Canada
shinhwei.tan@concordia.ca

ABSTRACT

Automated program repair aims to generate patches for buggy programs, a task often hindered by the cost of test executions in large projects. F1X introduces a novel methodology relying on test-equivalence relations, defining if two programs yield indistinguishable results for a specific test. By leveraging two test-equivalence relations based on runtime values and dependencies, F1X' algorithm categorises patches into test-equivalence classes, which helps to significantly reduce the number of required test execution to generate a patch without any information loss. Experiments on real-world programs from the ManyBugs benchmark demonstrated a substantial reduction in test executions, leading to efficiency gains over the previous methods, while retaining the patch quality. The efficiency and effectiveness of F1X was further shown in APR-COMP 2024, where it received the highest score in the Functional-C track.

CCS CONCEPTS

• **Software and its engineering → Automatic programming**; **Software testing and debugging**.

ACM Reference Format:
Sergey Mechtaev and Shin Hwei Tan. 2024. F1X at APR-COMP 2024. In *2024 ACM/IEEE International Workshop on Automated Program Repair (APR '24), April 20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3643788.3648018

1 INTRODUCTION

Automated program repair [1] is fundamentally the task of making precise alterations to a flawed program, denoted as P, to ensure it conforms to established correctness criteria, exemplified by a test suite T. This process results in a revised version of the program, referred to as P'. Program repair has attracted significant a attention of researches and practitioners that culminated in successful deployments, notably, at Meta [4] and Bloomberg [9].

Given the vast array of possible modifications (patches), navigating this extensive search space can be challenging. Specifically, test-driven program repair methods suffer from the high cost of test executions necessary to find a patch. To optmize this process, F1X, as outlined in Mechtaev et al. [5], categorizes potential patches into test-equivalence classes, efficiently reducing the number of required test executions by eliminating redundancies.

2 TEST-EQUIVALENCE ANALYSIS

The simplest algorithm of program repair, often referred to as generate-and-validate [7], enumerates and validates program modifications until it finds patch that satisfies the correctness criteria:

```
for candidate in Patches{
      P' = apply candidate to program P;
      check P’ against T;
      if P’ passes all tests in T
          break;
}
```

The main limitation of this algorithm is its scalability: since it enumerates changes one-by-one, it has to perform a large number of test execution, that are expensive in practice. As a result, it is able to explore only relatively small search spaces. In order to address this, we partition the space of programs in to test-equivalence classes.

Test equivalence. We call two programs P1 and P2 test-equivalent if they produce indistinguishable result on a given test. By establishing test-equivalence relations on the space of patches, it is possible to significantly reduce the number of validation steps, since it is sufficient to validate only a single patch from each test-equivalence class. We propose to use two test-equivalence analyses: value-based test-equivalence and dependency-based test-equivalence. Value-based test-equivalence indicates that two programs differ only on expression, and these expressions are evaluated into the same values during the test execution.

```
for (i=0; i<n; i++) {       for (i=0; i<n; i++){
  if (i % 2 == 1)               if (i == 1)
     print("1");                    print("1");
}                               }
```

For example, these two programs are test-equivalent for test `n=2`, since `i % 2 == 1` and `i == 1` are both evaluated into False, True during test execution.

Dependency-based test-equivalence captures programs that differ only in the locations at which an assignment statement is inserted in the source code, and this difference does not affect the data-flow during the test execution.

```
x = 1;                      if (n > 0){
if (n > 0) {                    x = 1;
    print(x);                   print(x);
}                           }
```

These two programs are test-equivalent for test `n=2`, since if-condition does not depend on the value of `x`.

3 RESULTS AND REFLECTION

Our method of identifying test-equivalence partitions of modifications reduces the number of required patch validation steps, and therefore help to repair bugs significantly faster. We compared with our algorithm and tool, F1X [5] with Angelix [6], GenProg [2] and

Prophet [3]. Our experiments show that F1X provides up to 10x speed-up on ManyBugs.

In APR-COMP Functional-C Track, there were three competing tools: F1X, Darjeeling and LLMR (based on ChatGPT [8]). F1X was the only tool that managed to generate patches that pass the private tests. Specifically, it generated patches for bugs in Libtiff and Grep applications. Here is an example of a generated patch for Grep:

```
--- a/src/dfasearch.c
+++ b/src/dfasearch.c
84c84
<       mp += dm->begline;
---
>       mp += eolbyte;
```

Despite the overall success of F1X in the competition, many patches it generated were incorrect, which remains a major concern in using test-driven program repair tools. We believe an integration of F1X with LLMs might alleviate it, since F1X is able to efficiently find a patch that passes the tests and LLM is able to reason about aspects of the problem related to natural language, such as the error message produced by the failing test, and take advantage of the historical patches observed in its training data. Thus, investigating a synergy of F1X and LLMs is a promising future direction.

4 COMPETITION ENTRY

F1X is an open source (MIT) project hosted on GitHub: https://github.com/mechtaev/f1x. It was originally developed by Sergey Mechtaev, Xiang Gao, Shin Hwei Tan and Abhik Roychoudhury, but later more researchers contributed to its implementation. The version submitted to the competition is also available on Zenodo: https://doi.org/10.5281/zenodo.8425412.

REFERENCES

[1] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[2] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72.

[3] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *POPL*. ACM, 298–312.

[4] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. Sapfix: Automated end-to-end repair at scale. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 269–278.

[5] Sergey Mechtaev, Xiang Gao, Shin Hwei Tan, and Abhik Roychoudhury. 2018. Test-equivalence analysis for automatic patch generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27, 4 (2018), 1–37.

[6] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *ICSE*.

[7] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA*. ACM, 24–36.

[8] John Schulman, Barret Zoph, Christina Kim, Jacob Hilton, Jacob Menick, Jiayi Weng, andCeron Felipe Juan Uribe, Liam Fedus, Luke Metz, Michael Pokorny, Rapha Gontijo Lopes, Shengjia Zhao, Arun Vijayvergiya, Eric Sigler, Adam Perelman, Chelsea Voss, Mike Heaton, Joel Parish, Dave Cummings, Rajeev Nayak, Valerie Balcom, David Schnurr, Tomer Kaftan, Chris Hallacy, Nicholas Turley, Noah Deutsch, Vik Goel, Jonathan Ward, Aris Konstantinidis, Wojciech Zaremba, Long Ouyang, Leonard Bogdonoff, Joshua Gross, David Medina, Sarah Yoo, Teddy Lee, Ryan Lowe, Dan Mossing, Joost Huizinga, Roger Jiang, Carroll Wainwright, Diogo Almeida, Steph Lin, Marvin Zhang, Kai Xiao, Katarina Slama, Steven Bills, Alex Gray, Jan Leike, Jakub Pachocki, Phil Tillet, Shantanu Jain, Greg Brockman, Nick Ryder, Alex Paino, Qiming Yuan, Clemens Winter, Ben Wang, Mo Bavarian, Igor Babuschkin, Szymon Sidor, Ingmar Kanitscheider, Mikhail Pavlov, Matthias Plappert, Nik Tezak, Heewoo Jun, William Zhuk, Vitchyr Pong, Lukasz Kaiser, Jerry Tworek, Andrew Carr, Lilian Weng, Sandhini Agarwal, Karl Cobbe, Vineet Kosaraju, Alethea Power, Stanislas Polu, Jesse Han, Raul Puri, Shawn Jain, Benjamin Chess, Christian Gibson, Oleg Boiko, Emy Parparita, Amin Tootoonchian, Kyle Kosic, and Christopher Hesse. 2022. Introducing ChatGPT. *OpenAI blog* (Nov 2022). https://openai.com/blog/chatgpt

[9] David Williams, James Callan, Serkan Kirbas, Sergey Mechtaev, Justyna Petke, Thomas Prideaux-Ghee, and Federica Sarro. 2023. User-Centric Deployment of Automated Program Repair at Bloomberg. *arXiv preprint arXiv:2311.10516* (2023).