ExpressAPR: Efficient Patch Validation for Java Automated Program Repair Systems

Yuan-An Xiao[†‡], Chenyang Yang[†‡], Bo Wang[§] and Yingfei Xiong[*†‡]
[†] Key Laboratory of High Confidence Software Technologies (Peking University), MoE, Beijing, China
[‡] School of Computer Science, Peking University, Beijing, China
[§] School of Computer and Information Technology, Beijing Jiaotong University, Beijing, China
{xiaoyuanan,chenyangy,xiongyf}@pku.edu.cn, wangbo_cs@bjtu.edu.cn

***Abstract*—Automated program repair (APR) approaches suffer from long patch validation time, which limits their practical application and receives relatively low attention. The patch validation process repeatedly executes tests to filter patches, and has been recognized as the dual of mutation analysis. We systematically investigate existing mutation testing techniques and recognize five families of acceleration techniques that are suitable for patch validation, two of which are never adapted to a general-purpose patch validator. We implement and demonstrate ExpressAPR, the first framework that combines five families of acceleration techniques for patch validation as the complete set. In our evaluation on 30 random Defects4J bugs and four APR systems, ExpressAPR accelerates patch validation for two order-of-magnitudes over plain validation or one order-of-magnitude over the state-of-the-art approach, benefiting APR researchers and users with a much shorter patch validation time.**

Demo video available at https://youtu.be/7AB-4VvBuuM

Tool repo (source code + Docker image + evaluation dataset) available at https://github.com/ExpressAPR/ExpressAPR

***Index Terms*—Automated Program Repair, Patch Validation, Mutation Analysis, Test Execution**

I. Introduction

Fixing software bugs is a tedious and costly procedure. Automated Program Repair (APR) aims to fix bugs without human intervention. APR has attracted much academic attention and is already deployed in large companies [1].

This paper focuses on APR efficiency, which is a relatively neglected aspect. The effectiveness of APR has been the main target of researchers: the ability to generate more correct patches is significantly improving. The most recent APR system is able to correctly fix 109 out of 391 real-world bugs [2]. In comparison, the efficiency of APR systems receives relatively low attention, but it is a vital factor affecting the practical usability of APR systems. In fact, state-of-the-art approaches still need up to several hours to fix a bug. For example, recent studies [3]–[5] set a 5-hour timeout for each bug, which is unaffordable for many users.

Existing APR approaches mostly follow the generate-and-validate fashion, i.e., they generate many patches and then validate the correctness of each patch against test cases. The efficiency problem is mainly caused by the patch validation step [6], which repeatedly executes tests. This scenario is similar to *mutation analysis*, to which APR has been recognized as the dual [7]. Accelerating mutation analysis has been deeply studied and some APR approaches already adapt accelerating techniques for patch validation. However, they only employ one or a few techniques and usually rely on ad-hoc implementation [8]. So far, there exist no systematic approaches that reveal the full acceleration potential.

We demonstrate ExpressAPR, the first tool that systematically combines five families of acceleration techniques as the complete set into a general-purpose patch validation framework. The five families are mutant schemata [9], mutant deduplication [10], test virtualization [11], test prioritization [12], and, parallelization. Users can configure them through options. The adaption is non-trivial due to the differences between mutation analysis and patch validation. In fact, the first two techniques have never been adapted to a general-purpose patch validator before ExpressAPR. We use a novel interception-based execution scheduling algorithm to enable the adaption, which is described in detail in our full paper [13].

The envisioned users of ExpressAPR are the researchers and users of Java APR systems. ExpressAPR provides out-of-the-box support for the Defects4J [14] benchmark and the Maven build tool. Therefore, researchers can easily use ExpressAPR to evaluate their approaches. ExpressAPR can also be customized to run on other Java projects, which benefits APR systems already deployed in the industry.

We evaluate ExpressAPR on 30 random Defects4J bugs with four APR approaches. It shows that ExpressAPR achieves an average acceleration of 108.9x over plain validation and outperforms the state-of-the-art approach by 10.3x, with nearly no impact on the precision of APR.

II. Background and Related Work

A. Acceleration Techniques for Mutation Analysis

Scalability is a key issue of mutation analysis [15] and researchers endeavor to propose accelerating approaches. We survey such existing techniques, from which we figure out five families of approaches that are suitable for APR patch validation and we also discuss the unsuitable ones.

Mutant Schemata: Standard mutation analysis compiles each mutant to an executable file separately, which introduces a huge compiling cost. Mutant schemata [9] alleviate the cost by replacing the mutated code location with a *sketch form program*. The sketch is implemented by a switch-case statement to select the mutant code snippet based on a runtime

[*] Corresponding author.

argument. In this way, we compile once to get a single executable file representing all mutants, reducing compiling costs.

Mutant Deduplication: Mutants are independently seeded at every possible location, and redundant mutants are common. A redundant mutant is semantically *equivalent* to either another mutant or the original program, or it produces the same output as them given a certain test input (*test-equivalent*). Equivalent mutants cannot be accurately detected for most programming languages. Researchers propose several lightweight filters, such as using compiler optimization to remove mutants leading to the same bytecode [16]. In comparison, test-equivalent mutants are more common for mutant deduplication approaches. For example, Major [17] identifies test-equivalent mutants by a pre-pass that records the values of mutants' operands and analyzes each mutant's state.

Test Virtualization: Java programs are executed on a Virtual Machine (VM), which suffers from a high initialization cost. Given the same test input and the same environmental settings, all mutants share the same initialization processes. Test virtualization techniques speed up execution by reducing initialization redundancies. VMVM [11] reuses a booted JVM across tests by resetting its modified states.

Test Case Prioritization: In the scenario of mutation analysis, killing a mutant (i.e., a test fails on a mutant) earlier can avoid executing the remaining tests. Researchers propose to prioritize test cases by their coverage [12], so as to enlarge the probability of killing mutants. The patch validation procedure of APR meets the same requirement, and APR systems commonly use heuristic prioritizing policies, e.g., executing the original failed test cases first.

Parallelization: Because tests and mutants are naturally independent of each other, it is intuitive to parallelize testing by separating tests or mutants into multiple parallel processes.

Other unsuitable techniques: Some accelerating techniques need mechanisms that are unable to access from Java programs. For example, fork-based mutation analysis [18]–[20] needs the fork mechanism, which is missing in Java.

B. Accelerating Patch Validation

There exists a few studies to accelerate patch validation, which is essentially adapting corresponding techniques from mutation analysis. UniAPR [8] is the state-of-the-art patch validation acceleration framework based on test virtualization. Similarly, Guo et al. [21] propose to accelerate by dynamic software updating. PraPR [22] directly modifies Java bytecode to avoid compilation costs. Mechtaev et al. [23] propose two test-equivalence relations to filter patches. They only adapt one certain technique, and some studies depend on a specific APR approach while ExpressAPR supports arbitrary patches.

III. ExpressAPR Overview

A. The Methodology

The core of ExpressAPR is an automated process of source-code level instrumentation that realizes the discussed acceleration. This core process runs under JDK 1.8+ and works for all Java projects with JUnit tests under the source level of 1.5+. We further provide a command-line interface as a high-level interface, which handles periphery details such as classpaths and error recovery.

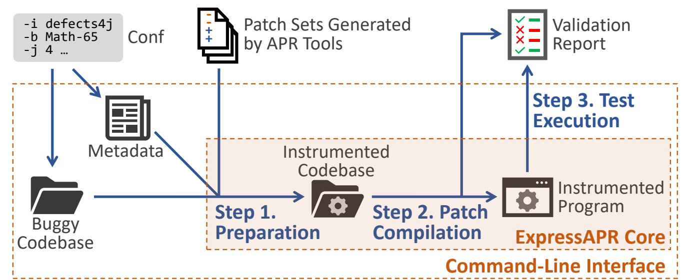

Fig. 1. Workflow of ExpressAPR

Figure 1 demonstrates the workflow of using ExpressAPR. The input is the **patch sets** to be validated, and some **configuration** flags describing the project under validation and used acceleration techniques. The command-line interface then prepares the environment and starts multiple worker processes in parallel, each process automatically calling the ExpressAPR Core to validate a set of patches in below three steps:

1) **Preparation**, where it instruments the codebase to weave all patches into it, forming mutant schemata. The runtime procedure for test execution is also instrumented into the codebase. 2) **Patch Compilation**, where the instrumented codebase is compiled into an executable program. In this way, all patches are compiled at once, saving compilation time. Should there be compile errors, ExpressAPR parses the error message to remove all patches leading to errors from the codebase (reporting them as uncompilable) and compile the codebase again, which should be successful. 3) **Test Execution**, where the instrumented runtime procedure is executed to run the test suite. Mutant deduplication, test virtualization, and test case prioritization are all included in the runtime procedure. Finally, ExpressAPR collects the validation result into a validation report file.

Note that adapting accelerating approaches for mutation analysis can be non-trivial due to the differences between mutation analysis and patch validation. In mutation analysis, all mutants are generated from *pre-defined* mutation operators, whereas in patch validation, patches are *arbitrary* changes to statements that are not under our control. We propose an interception-based execution scheduling algorithm in ExpressAPR Core to enable acceleration of arbitrary statement-level patches. Interested readers could refer to our full paper [13].

B. Input and Output Format

As a general-purpose patch validator, ExpressAPR does not depend on a specific APR approach. Instead, it interfaces with any APR approach with patch set files. A patch set file is a JSON manifest that describes all patches modifying the same part in the program to be validated. For example, below is a manifest file containing three patches:

```
{ "manifest_version": 3,
  "interface": "defects4j", "bug": "Math-65",
```

```
"filename": "src/main/java/Modified.java",
"context_above": "public class Modified {
   void foo() { int x =",
"unpatched": "Integer.MAX_VALUE",
"context_below": "; int y = x + 1; } }",

"patches": [
  "-100", "-100; x++", "Integer.MAX_VALUE; return"
] }
```

The first three fields, `manifest_version`, `interface`, and `bug`, describe the format of this file (currently v3) and the Bug ID of the project under repair (Math-65 in the Defects4J benchmark). These fields are for error-proof purposes, so that ExpressAPR will issue a warning for possible inconsistency. The middle four fields describe the location of patches, such that the original content of `filename` is `context_above + unpatched + context_below`, where the `unpatched` part will be replaced by each patched version described in the `patches` field. We do not require patches to be aligned to a statement boundary. ExpressAPR works fine even if the `unpatched` part is a part of the statement or multiple statements, as long as it does not span multiple methods.

The output of ExpressAPR is a validation report file containing the test result of each validated patch. The file is in JSONL format, where each line is encoded as JSON and corresponds to an input patch set. Here is an example (prettified) line:

```
{ "patches_path": "/path/to/patch_set.json",
  "succlist": "FFs",
  "technique": "expapr",
  "extra": {...} }
```

The `patches_path` field is the filename of the input patch set. The `succlist` field is a string, the i-th character describing the validation result of the i-th patch in this patch set, which can be one of the following characters:

- **C** ("compile error"): the patch does not compile;
- **F** ("failed"): the patch compiles, but a test case fails or does not terminate within the timeout[1].
- **s** ("success"): the patch successfully compiles and passes all test cases in the project;

The `technique` field can be either `"expapr"`, indicating that this patch set is successfully accelerated by ExpressAPR, or `"fallback"`, indicating that it is beyond our capability (e.g., the patch location spans multiple methods) and ExpressAPR falls back to plain validation for this patch set. The `extra` field contains extra diagnosis information, such as the reason when `technique` is `"fallback"`.

C. The Command-Line Usage

While the user can directly use the low-level ExpressAPR Core to carry out the patch validation step, the command-line interface (CLI) is the desired way to use ExpressAPR. It provides out-of-the-box support for the Defects4J [14] benchmark, so that APR researchers can switch to ExpressAPR with two simple commands. Here we demonstrate its usage.

The user first initializes the project under test into an empty working directory with this command:

```
> expapr-cli init -i defects4j -b Math-65
    -w /tmp/workdir -j 4 -d trivial
```

- `-i defects4j` and `-b Math-65` specify the project to validate;
- `-w /tmp/workdir` specifies the working directory to initialize;
- `-j 4` enables parallel patch validation with 4 processes;
- `-d trivial` turns on basic mutant deduplication[2].

The initialization step checkouts the project and executes the test suite to collect test case prioritization information and calculate test timeout. This step only happens once per project and can be performed before repair.

The user then runs an APR tool that generates candidate patches in the patch set format described above. Most APR approaches generate patches by iterating the locations returned by a fault localization approach. In this case, candidate patches naturally form patch sets based on their locations.

Finally, the user runs this command to validate the patches:

```
> expapr-cli run -w /tmp/workdir
    "/tmp/patches/*.json"
```

- `-w /tmp/workdir` specifies the working directory, as initialized by the `init` command;
- `"/tmp/patches/*.json"` specifies the glob pattern to find the patch sets to be validated;
- for diagnosis purposes, test case prioritization can be disabled with `--no-prio`; mutant deduplication can be disabled with `--no-dedup`; runtime acceleration techniques can be totally disabled with `-t fallback`.

The validation report will be stored in the working directory with filename `result.jsonl`. Because the report file is line-based, the patch validation process can be interrupted at any time (e.g., when the first correct patch is found), leaving the file incomplete. Re-running the `expapr-cli run` command will continue this process.

D. Supporting Other Projects

ExpressAPR is compatible with various Java projects through an interface for checking out the project, getting the classpath for compiling it, and running the plain test validation process (as a fallback). While the CLI script provides out-of-the-box support for Defects4J (`-i defects4j -b Math-65`) and Maven (`-i maven -b /path/to/project`), it can also run on other projects by implementing the methods in the Python class `Interface` defined in `interface/__init__.py`.

We plan to add built-in support for other well-known benchmarks and build systems in the future.

IV. Empirical Evaluation

We conducted an empirical evaluation to understand ExpressAPR's performance over UniAPR [8], the state-of-the-art Java patch validator, and the plain patch validation command from Defects4J (`defects4j compile && defects4j test`).

[1]The timeout is customizable. Its default value is 5 seconds plus 1.5 times the original test execution time, following an existing study [6].

[2]Mutant deduplication relies on an external database of methods that are known to be side-effect free. `-d trivial` picks the built-in database of well-known pure methods such as `toString`.

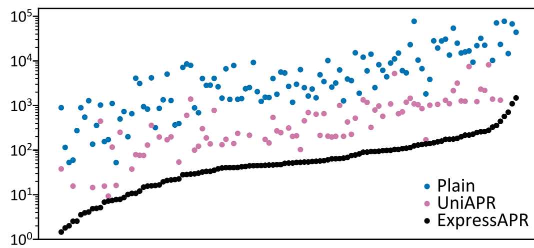

Fig. 2. Patch validation time (seconds) per bug

We chose four representative APR tools to be studied in our evaluation, namely Recoder [3], TBar [24], SimFix [5], and Hanabi [25]. We run each tool against 30 randomly chosen bugs in Defects4J v1.2, with slight modifications so that the tool will store candidate patches onto the disk under the format required by ExpressAPR. Then, we used ExpressAPR and the two baseline approaches to validate collected patches. Please refer to our full paper [13] for additional experiment settings.

RQ1: Effectiveness. Figure 2 shows the total patch validation time per bug with different approaches[3]. We can see that for all four APR tools and all 30 bugs, ExpressAPR is the fastest approach. ExpressAPR is averagely **108.9x faster over plain validation** or **10.3x faster over UniAPR** for these bugs. With ExpressAPR, the patch validation generally takes no more than one or a few minutes per bug on a mainstream computer, which is no longer the time bottleneck of APR.

RQ2: Feasibility. We compare the validation report of ExpressAPR with the ground truth from the plain baseline. It shows that ExpressAPR **successfully validates 98.782% of patches** with the correct result. 1.217% of patches are beyond the capability of ExpressAPR so the command-line script automatically falls back to the plain approach. **Only the remaining 0.001% of patches show an incorrect result** (reported as "`s`" where it should be "`C`" or "`F`", or vice versa). Therefore, the feasibility of ExpressAPR is high, and it has minimal impact on the precision of APR.

V. Conclusion

We have presented ExpressAPR, a framework for accelerating the patch validation of Java APR systems. ExpressAPR systematically adapts five families of accelerating techniques as the complete set. The tool contains a user-friendly interface and can be customized. The evaluation results show that ExpressAPR significantly accelerates the patch validation procedure compared with the state-of-the-art technique.

Acknowledgment

This work was supported by the Fundamental Research Funds for the Central Universities (2022RC033), the National Natural Science Foundation of China (62202040), the China Postdoctoral Science Foundation (2021M700367), and a grant from ZTE-PKU Joint Laboratory for Foundation Software.

[3]Note that some points for the UniAPR series are missing because it fails to run on 27% of bugs. UniAPR is implemented for Maven, which is not used in some Defects4J projects. While we ported some projects to Maven, UniAPR still fails on some projects due to dependency problems.

References

[1] S. Kirbas, E. Windels, O. McBello, K. Kells, M. Pagano, R. Szalanski, V. Nowack, E. R. Winter, S. Counsell, D. Bowes, T. Hall, S. Haraldsson, and J. Woodward, "On the introduction of automatic program repair in Bloomberg," *IEEE Softw.*, vol. 38, no. 4, pp. 43–51, Jul. 2021.
[2] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proc. ICSE*, 2023.
[3] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proc. ESEC/FSE*, 2021, p. 341–353.
[4] Y. Li, S. Wang, and T. N. Nguyen, "DLFix: Context-based code transformation learning for automated program repair," in *Proc. ICSE*, 2020, pp. 602–614.
[5] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proc. ISSTA*, 2018, pp. 298–309.
[6] A. Ghanbari and A. Marcus, "PRF: A framework for building automatic program repair prototypes for JVM-based languages," in *Proc. ESEC/FSE*. New York, NY, USA: ACM, Nov. 2020, pp. 1626–1629.
[7] W. Weimer, Z. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Proc. ASE*, 2013, pp. 356–366.
[8] L. Chen, Y. Ouyang, and L. Zhang, "Fast and precise on-the-fly patch validation for all," in *Proc. ICSE*. IEEE, 2021, pp. 1123–1134.
[9] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *Proc. ISSTA*, 1993, pp. 139–148.
[10] R. Just, F. Schweiggert, and G. M. Kapfhammer, "MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler," in *Proc. ASE*, 2011, pp. 612–615.
[11] J. Bell and G. Kaiser, "Unit test virtualization with VMVM," in *Proc. ICSE*, 2014, pp. 550–561.
[12] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *Proc. ISSTA*, 2013, pp. 235–245.
[13] Y.-A. Xiao, C. Yang, B. Wang, and Y. Xiong, "Accelerating patch validation for program repair with interception-based execution scheduling," 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.03955
[14] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for java programs," in *Proc. ISSTA*. New York, New York, USA: ACM Press, 2014, pp. 437–440.
[15] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.
[16] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *Proc. ICSE*, 2015, pp. 936–946.
[17] R. Just, M. D. Ernst, and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states," in *Proc. ISSTA*, 2014, pp. 315–326.
[18] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing," *Software: Practice and Experience*, vol. 21, no. 7, pp. 685–718, 1991.
[19] B. Wang, Y. Xiong, Y. Shi, L. Zhang, and D. Hao, "Faster mutation analysis via equivalence modulo states," in *Proc. ISSTA*. ACM, 2017, pp. 295–306.
[20] B. Wang, S. Lu, Y. Xiong, and F. Liu, "Faster mutation analysis with fewer processes and smaller overheads," in *Proc. ASE*. IEEE, 2021, pp. 381–393.
[21] R. Guo, T. Gu, Y. Yao, F. Xu, and X. Ma, "Speedup automatic program repair using dynamic software updating: an empirical study," in *Proc. Internetware*, 2019, pp. 1–10.
[22] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proc. ISSTA*, 2019, pp. 19–30.
[23] S. Mechtaev, X. Gao, S. H. Tan, and A. Roychoudhury, "Test-equivalence analysis for automatic patch generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 4, pp. 1–37, 2018.
[24] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBar: revisiting template-based automated program repair," in *Proc. ISSTA*, 2019, pp. 31–42.
[25] Y. Xiong and B. Wang, "L2S: A framework for synthesizing the most probable program under a specification," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 3, 2022.