PRF: A Framework for Building Automatic Program Repair Prototypes for JVM-Based Languages

Ali Ghanbari
University of Texas at Dallas
Richardson, TX 75080, USA
ali.ghanbari@utdallas.edu

Andrian Marcus
University of Texas at Dallas
Richardson, TX 75080, USA
amarcus@utdallas.edu

ABSTRACT

PRF is a Java-based framework that allows researchers to build prototypes of test-based generate-and-validate automatic program repair techniques for JVM languages by simply extending it with their patch generation plugins. The framework also provides other useful components for constructing automatic program repair tools, *e.g.,* a fault localization component that provides spectrum-based fault localization information at different levels of granularity, a configurable and safe patch validation component that is 11+X faster than vanilla testing, and a customizable post-processing component to generate fix reports.
A demo video of PRF is available at https://bit.ly/3ehduSS.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Automatic Program Repair, Framework, Fault Localization, Patch Validation

ACM Reference Format:
Ali Ghanbari and Andrian Marcus. 2020. PRF: A Framework for Building Automatic Program Repair Prototypes for JVM-Based Languages. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3368089.3417929

1 INTRODUCTION

Automatic program repair (APR) [8] is one of the recent advances in automated software engineering that holds the promise of reducing debugging costs by suggesting high-quality patches that either directly fix the bugs or help the developers during manual debugging [12]. In the last decade, APR has been the subject of intense research and still remains an active area of research [4, 16].

Generate-and-validate (G&V) refers to the class of APR techniques that attempt to fix the bug by repeatedly generating patches to produce program variants that subsequently get validated against certain rules or checks. A patch is called *plausible* if it passes all the checks. Validating a patch can be accomplished via a spectrum of techniques ranging from sound formal verification techniques [20] to testing [6, 14, 21]. However, in real-world situations, formal specification of software is usually absent and automating formal verification is theoretically impossible. Testing, on the other hand, remains as the prevalent, economic method of getting more confidence about the quality of software, and a vast majority of G&V APR techniques use tests as correctness criteria.

By examining the architecture of state-of-the-art test-based G&V APR tools [4], we observed that most of them have components for fault localization (FL) [23], patch generation, and patch filtering/prioritization. During fault localization a *suspiciousness value* is assigned to each executed program location. During patch generation, a subset of these locations are transformed, resulting in a set of patches. Since test cases do not perfectly specify the software, many of the generated patches happen to pass all the tests without fixing the bugs. Such patches are known as *overfitted* patches [19]. Therefore, the APR process is usually followed by a post-processing phase to filter out overfitted patches or prioritize patches that are more likely to be correct. Different program repair techniques usually differ in the way they generate patches to produce program variants, while the other components remain more or less the same and are reused from one implementation to another or reinvented. Even reusing these processes takes a considerable amount of programming effort for the APR researchers when building their prototypes.

This paper presents PRF (**P**rogram **R**epair **F**ramework), a framework for building APR prototypes by extending it with a patch generation plugin for transforming the identified suspicious locations. PRF provides the APR tool programmer with spectrum-based FL information at different levels of granularity (*e.g.,* line-/statement-, method-, or class-level). The framework also provides a safe, fast, and configurable facility for patch validation, and a customizable fix report generation component for prioritization/filtering of plausible patches. The patch validation component employs an assortment of techniques to safely accelerate patch validation process; it reorders test cases and does test selection. The component also implements a novel work-stealing algorithm [22] so that the patches can be validated concurrently, thereby maximizing CPU utilization. This compensates for the overhead of repeated creation of Java Virtual Machine (JVM) processes to achieve safety in patch validation. Although PRF is shipped with a default fix report generation plugin, the APR tool programmers can construct their own patch prioritization/filtering plugins for a customized fix report generation.

PRF depends solely on runtime information, so it is useful not just for prototyping Java APR tools but also for building tools targeting other JVM-based programming languages such as Kotlin, Scala, etc.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3417929

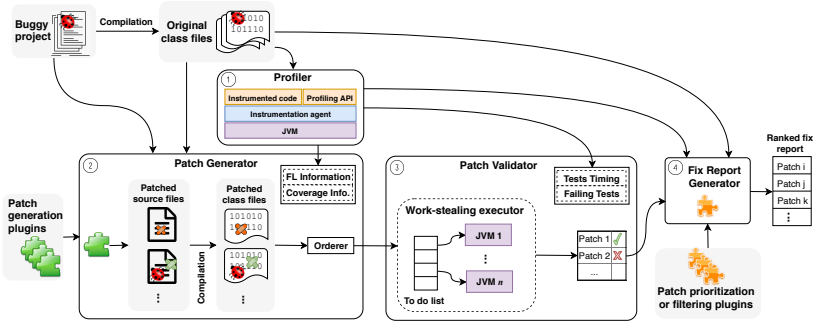

Figure 1: An overview of PRF

It also comes in the form of a Maven plugin so that it can be reliably applied on different Maven-based projects.

We have constructed a patch generation plugin by specializing CapGen APR tool proposed by Wen *et al.* [21]. Our experiments with CapGen show that patch validation is responsible for 77.7% of the total repair time, on average, and our integration with PRF results in 3.1X speedup when reordering test cases while validating patches sequentially without any test selection. In addition, we obtained an 11.4X speedup in patch validation when using 8 CPU cores combined only with test reordering.

Constructing a plugin for CapGen is as simple as making a wrapper for the tool and disabling the patch validation and prioritization processes that are to be delegated to PRF. We expect that with the help of APR libraries like ASTOR [14], APR researchers will be able to make fast and reliable prototypes easily. This will, in turn, greatly help the reproducibility of APR experiments. Such a customizable framework has the added benefit of simplifying studies that compare different algorithms used in certain aspects of APR (*e.g.*, fault localization [13] or patch classification/prioritization [24]).

PRF and its documentation, together with experimental result for CapGen, are publicly available on GitHub [7].

2 OVERVIEW OF PRF

Inspired by the steps common to state-of-the-art test-based G&V APR techniques, we build PRF with four components responsible for profiling, patch generation, patch validation, and fix report generation. Figure 1 shows PRF's main components. We next describe each component in more details.

2.1 Profiler

Any non-trivial G&V APR tool needs information about the program under repair so as to make sensible transformations, validate the generated patches efficiently, or to generate high-quality fix reports. For example, in most cases, it makes sense to mutate only suspicious program locations, and in order to achieve performance in patch validation the tool might reorder test cases based on their execution time. We use the term *profiling* to refer to the task of obtaining information about the program under repair. These include test results and execution time, coverage information, dynamic call graph, and fault localization information that are used in different components for various purposes such as patch generation, test selection, test reordering, and patch prioritization.

Since APR tool programmer, or the end users, might opt for other sources for these information, except for test execution time measurement and recording test results, which becomes freely available once PRF executes the tests, all features of the *profiler* component are optional.

During profiling, the program is instrumented for recording the aforementioned runtime information. PRF uses Java Agent technology and the ASM bytecode manipulation framework [17] to instrument the input program. Besides constructing dynamic call graph and recording test results, test execution time and, depending on the user configuration, coverage information of test cases at different levels of granularity (*i.e.*, class-, method-, or line-level) are recorded. If the user selects a certain level of granularity and specifies a certain FL formula [23], then the collected coverage information is used to calculate the suspiciousness values for the program elements based on the specified formula.

2.2 Patch Generator

Patch generation is an integral part of G&V APR algorithms wherein the patches are constructed via transforming a subset of program locations. Different APR algorithms mostly differ in the way they generate the patches, and the philosophy behind PRF is to separate patch generation phase from the rest of phases and create a generic, customizable program repair framework.

In PRF, the *patch generator* component is responsible for generating patches which is customizable via a user provided *patch generation plugin*. Depending on the patch generation algorithm, the program source code and/or compiled class files of the original buggy program are transformed by the specified patch generation plugin. Thus, the framework passes the path name of the source files, test sources, and compiled binaries to the plugin. Additionally, depending on which feature of the profiler component are activated, test coverage information, dynamic call graph, and FL information with the specified level of granularity shall be fed to the plugin.

The patch generation plugin is intended to generate a pool of patches that are stored on the disk. PRF expects the patches to be compiled into class files and it is the responsibility of the patch generation plugin to compile the generated patches using the appropriate JVM-based compiler. Furthermore, in order to do test selection in patch validation phase, PRF relies on the patch generation plugin to determine which test cases cover the patched location for each patch. If the plugin does not specify the covering tests for a given patch, then during the subsequent patch validation phase, all the test cases shall be executed against the patch and no test selection will take place.

2.3 Patch Validator

The patch generator component sends the list of generated patches to the *patch validator* component wherein (a subset of) the test cases are executed against the patches to identify plausible patches. Depending on the degree of parallelism specified by the user, the component validates the patches sequentially or in parallel. Furthermore, depending on the user preferences, patch validation can be stopped after the first plausible patch is found, or the entire search space will be explored and the identified plausible patches will get prioritized and/or filtered later.

It is worth noting that patch validation takes up a large portion of the end-to-end repair time. For example, Figure 2 shows the time that an existing APR tool, CapGen [21], spends on generating patches vs. on validating the generated patches, for each of the 22 Defects4J [10] bugs that it is able to fix. Our measurements indicate that patch validation takes up 77.7% of total repair time, on average. Similar observations are also reported in previous research by Mehne et al. [15]. As we shall discuss in §4, there are a number of methods to reduce patch validation time, but in this work we employ a novel approach: (1) each patch is validated in a separate process; (2) only the tests covering patched location are executed; (3) unless otherwise instructed by the user, PRF reorders test cases to run shorter tests first and also originally failing tests before originally passing ones; (4) unless otherwise instructed by the user, PRF runs patch validating processes in parallel.

Since patch validation involves running test cases and test execution have side-effects, PRF validates each patch in a separate JVM instance to contain the side-effects as much as possible. Repeated creation of JVM instances is expensive as initializing a JVM session, besides process creation overhead, involves costly tasks of class loading, linking, and JIT-optimization. These costs are compensated for by the speedup gained through test selection, test reordering, and parallelism.

By using test selection and test reordering methods from regression testing, which have been successfully applied in APR [6, 15], PRF runs only the test cases that cover the patched locations and runs the failing test cases before the passing ones as they are more likely to fail again and invalidate the incorrect patch as soon as possible. PRF aims for further speeding up patch validation process by overlapping JVM process creation with the effective work done during patch validation phase, namely running test cases. This is achieved via an implementation of work-stealing algorithm [22] which ensures CPU utilization is maximized by always keeping its cores busy doing different tasks in parallel.

Sometimes patching may create infinite loops and this causes tests to run forever during patch validation. To avoid this issue, PRF uses test execution time recorded during profiling phase to identify patches that are likely to have entered an infinite loop, by setting a timeout for each test case. We use the formula $\beta + (1 + \alpha)\tau_t$ to calculate the time budget for a test t, based on the original execution time τ_t of the test and the user-defined parameters α and β. Following heuristics determined during our experiments, we set $\beta = 5,000$ and $\alpha = 0.5$, meaning that a test taking more than 1.5 times its original execution time plus 5 seconds is deemed timed out. A patch for which at least one test times out, is deemed non-plausible and will not be passed to the next component.

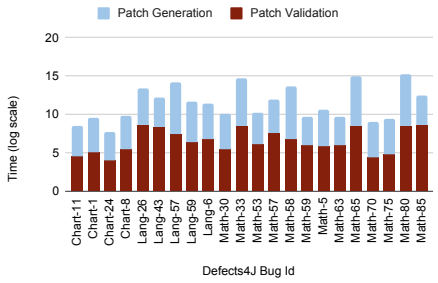

Figure 2: Patch generation time for CapGen and validation time for the generated patches, for 22 bugs in Defect4J.

2.4 Fix Report Generator

The *fix report generator* component produces a list of patches to be examined by the user of APR techniques. It may use a patch prioritization and/or filtering plugin to first display the patches that are more likely to be correct or filter out plausible but likely incorrect patches. Such a plugin would be provided by the user. The current implementation of the fix report generator prints the list of plausible patches in an arbitrary order which might be time consuming and boring. We plan to integrate our JVM language agnostic patch prioritization tool ObjSim [5], which has shown promising results in our experiments with the APR tool PraPR [6].

3 PRF USAGE

PRF comes in the form of a Maven plugin. After checking out the source code from the GitHub repository [7] and installing the Maven plugin and the core library for PRF on the local Maven repository, the framework will be ready to use. The following XML snippet shows the minimal amount of configuration in the POM file of the target buggy project.

```
<plugin>
    <artifactId>prf-maven-plugin</artifactId>
    <groupId>edu.utdallas</groupId>
    <version>1.0-SNAPSHOT</version>
</plugin>
```

Once `mvn edu.utdallas:prf-maven-plugin:run` is executed through the command-line, PRF shall use a default patch generation plugin named `DummyPatchGenerationPlugin` which simply looks for a directory named `patches-pool` under the base directory of the project. This directory is expected to contain the pool of the generated patches, and the class file(s) for each patch must reside in a separate sub-directory. Names of the sub-directories shall be used as patch identifiers during the fix report generation. Similarly, PRF uses a default patch prioritization plugin named `DummyPatchPrioritizationPlugin` that does not have any effect. Furthermore, by default, only test execution time and test result collector is active in the profiler, and patch validator uses all the available CPU cores to validate patches in parallel.

These default choices can be overridden by the user through the POM file for the program under repair. Such configurations go under the `<configuration>` tag in plugin description in the above XML code. Table 1 summarizes the options that the users can tune. More details about using PRF is available in the project documentation [7] and in the companion demo video https://bit.ly/3ehduSS.

Table 1: Summary of PRF options configurable through the POM file

Option	Descriptions
<flOptions>	Takes values OFF for no FL, or CLASS_LEVEL, METHOD_LEVEL, or LINE_LEVEL for class-/method-/line-level FL
<flStrategy>	Takes the values OCHIAI or TARANTULA, for Ochiai or Tarantula spectrum-based FL [23], respectively
<testCoverage>	If true, line-level coverage information for tests will be collected and passed to other components
<failingTests>	If left empty, failing tests will be inferred. Each failing should be in a <failingTest> tag
<cgOptions>	Takes values OFF (default) or DYNAMIC to enable/deactivate dynamic call graph construction
<patchGenerationPlugin>	The name of patch generation plugin. By default, this is set to dummy-patch-generation-plugin
<parallelism>	Degree of parallelism for patch validation. By default, it is 0, and all CPU cores will be used
<patchPrioritizationPlugin>	The name of patch prioritization plugin. By default, this is set to dummy-patch-prioritization-plugin
<timeoutConstant>	Constant part of the timeout value calculation, *i.e.,* β in §2.3
<timeoutPercent>	Percent part of the timeout value calculation, *i.e.,* α in §2.3

4 RELATED WORK

FLAIR is a proprietary framework used in Fujitsu Labs America to construct the APR system ELIXIR [18]. However, this system is not publicly available. PRF is the first open-source framework that provides all the functionalities offered by a framework like FLAIR, plus PRF offers an efficient patch validation facility.

ASTOR [14] is a general-purpose library for developing source code level, Java-based APR tools. Unlike ASTOR, PRF is a customizable framework upon which different patch generation algorithms can be installed as plugins and different strategies for patch prioritization and/or filtering can be employed. PRF is not intended to replace ASTOR, instead, PRF complements the library in that it enables APR researchers focus on taking full advantage of ASTOR's features to implement more reliable patch generation algorithms, and better understand the impact of different fault localization, patch prioritization and/or filtering strategies on the effectiveness of their implementation. As a toolset for prototyping APR techniques, PRF can be seen as a step similar in nature to [3].

Le Goues *et al.* [11] highlight the high cost of patch validation in test based G&V APR. Mehne et al. [15] report that patch validation can take between 40% to 92% of total repair time and propose to prune the patches needed to be tested as well as test case selection to reduce this cost. A recent line of research [6, 9], proposes to use the HotSwap trick offered by the JVM to validate the patches on-the-fly, without restarting the JVM. This was previously used in mutation testing systems like PIT [2]. It has the benefit of avoiding the high costs of restarting JVM for each patch, but the patches that involve altering the class structure (*e.g.,* addition or removal of class members) are not eligible for being HotSwapped. The approach in order to be effective needs proper isolation of test execution side-effects (*e.g.,* [1]). However, such a method does not solve the current limitations of a HotSwap-based approach. In this work, we follow a different approach that proves to be fast, reliable, and more effective. PRF validates each patch in a fresh JVM session, thereby containing the side-effects of test execution and also avoiding restrictions of a HotSwap-based approach, and instead of dealing with the internals of JVM, we rely on test selection and reordering, as well as parallelism to speedup patch validation.

5 CONCLUSIONS AND FUTURE WORK

With PRF, APR researchers will be able to build research prototypes by simply providing a patch generation plugin and fine tuning the framework's existing components for multi-granularity level fault localization, patch validation, and fix report generation. PRF uses a novel patch validation technique, relying on test selection and prioritization as well as parallelism, that achieves 11+X speedup compared to vanilla testing. PRF is publicly available at [7].

We are working on integrating our JVM language agnostic patch prioritization system ObjSim [5] in PRF and release the framework with an effective built-in patch prioritization mechanism.

REFERENCES

[1] Jonathan Bell and Gail Kaiser. 2014. Unit test virtualization with VMVM. In *ICSE*.
[2] Coles, Henry. 2020. PIT Mutation Testing. http://pitest.org/ Accessed: 06/20.
[3] Gregory Gay and Rene Just. 2020. Defects4J as a Challenge Case for the Search-Based Software Engineering Community. In *SSBSE*. to appear.
[4] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic software repair: A survey. *TSE* (2017).
[5] Ali Ghanbari. 2020. ObjSim: Lightweight Automatic Patch Prioritization via Object Similarity. In *ISSTA*. 541–544.
[6] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical Program Repair via Bytecode Mutation. In *ISSTA*.
[7] Ali Ghanbari and Andrian Marcus. 2020. https://bit.ly/37z5RES. Accessed: 06/20.
[8] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *CACM* (2019).
[9] Rongxun Guo, Tianxiao Gu, Yuan Yao, Feng Xu, and Xiaoxing Ma. 2019. Speedup Automatic Program Repair Using Dynamic Software Updating: An Empirical Study. In *APSI*. 1–10.
[10] René Just, Darioush Jalali, and Michael Ernst. 2014. Defects4J. https://bit.ly/2PY3yDa Accessed: 06/20.
[11] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current challenges in automatic software repair. *SQJ* (2013), 421–443.
[12] Jingjing Liang, Jiru Yi, Jiajun Jiang, YIling Lou, Yingfei Xiong, and Gang Huang. 2020. Interactive Patch Filtering as Debugging Aid. *arXiv preprint arXiv:2004.08746* (2020).
[13] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *ICST*. 102–113.
[14] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java (Demo). In *ISSTA*.
[15] Ben Mehne, Hiroaki Yoshida, Mukul R Prasad, Koushik Sen, Divya Gopinath, and Sarfraz Khurshid. 2018. Accelerating search-based program repair. In *ICST*. 227–238.
[16] Martin Monperrus. 2018. *The Living Review on Automated Program Repair.* Technical Report hal-01956501. HAL/archives-ouvertes.fr.
[17] OW2 Consortium. 2020. ASM. https://bit.ly/3fsPL2r Accessed: 06/20.
[18] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. Elixir: Effective object-oriented program repair. In *ASE*. 648–659.
[19] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *FSE*.
[20] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *ICSE*. 151–162.
[21] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *ICSE*.
[22] Wikipedia contributors. 2020. Work stealing — Wikipedia, The Free Encyclopedia. https://bit.ly/2XZEhwc Accessed: 06/20.
[23] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *TSE* (2016), 707–740.
[24] He Ye, Matias Martinez, and Martin Monperrus. 2019. Automated Patch Assessment for Program Repair at Scale. *arXiv* (2019).