ObjSim: Lightweight Automatic Patch Prioritization *via* Object Similarity

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

ABSTRACT

In the context of test case based automatic program repair (APR), patches that pass all the test cases but fail to fix the bug are called *overfitted* patches. Currently, patches generated by APR tools get inspected manually by the users to find and adopt genuine fixes. Being a laborious activity hindering widespread adoption of APR, automatic identification of overfitted patches has lately been the topic of active research. This paper presents engineering details of ObjSim: a fully automatic, lightweight similarity-based patch prioritization tool for JVM-based languages. The tool works by comparing the system state at the exit point(s) of patched method before and after patching and prioritizing patches that result in state that is more similar to that of original, unpatched version on passing tests while less similar on failing ones. Our experiments with patches generated by the recent APR tool PraPR for fixable bugs from Defects4J v1.4.0 show that ObjSim prioritizes 16.67% more genuine fixes in top-1 place. A demo video of the tool is located at https://bit.ly/2K8gnYV.

CCS CONCEPTS

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

KEYWORDS

Automatic Program Repair, Patch Prioritization, Object Similarity, Test Case

ACM Reference Format:
Ali Ghanbari. 2020. ObjSim: Lightweight Automatic Patch Prioritization *via* Object Similarity. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3395363.3404362

1 INTRODUCTION

Manual debugging is notoriously difficult and costly. Automatic program repair (APR) [11] aims to reduce the costs by suggesting high-quality patches that either directly fix the bugs or help the human developers during manual debugging. Generate-and-validate (G&V) refers to the class of APR techniques that attempt to fix the bug by first generating a pool of patches and validating the patches via certain rules and/or checks. A patch is said to be *plausible* if it passes all the checks. Ideally, we would apply a sound method (e.g., formal verification) for checking validity of generated patches. However, in real-world situations, formal specifications of software are usually absent and due to theoretical limitations, formal verification is in general not automatable. In contrast, testing is the prevalent, economic method of getting more confidence about the quality of software. So, a vast majority of G&V APR techniques, indeed almost all APR techniques, use test cases as correctness criteria for patches [7].

A test-based G&V APR algorithm receives as input a buggy program and, optionally, a test suite consisting of at least one failing test identifying the bug, and produces zero or more plausible patches, i.e., patches that pass all the tests. A typical APR process in this class starts with fault localization to locate likely faulty program locations. It then attempts to fix the bug by applying a number of transformation operators on the identified suspicious locations to obtain a pool of candidate patches to be tested against the existing test suite. Patches that pass all the test cases are reported as plausible patches.

Unfortunately, test cases only partially specify the behavior of the programs and many of the generated patches happen to pass all of the test cases without actually fixing the bug. This makes APR techniques produce many plausible but incorrect patches, aka *test case overfitted* patches [17] (or simply overfitted patches). The process of examining APR-generated patches has to be manual for the oracle problem is undecidable, but in some cases, manually analyzing each and every one of the plausible patches could be even costlier than directly fixing the bug [17]. Thus, a convenient method for post-processing the generated patches before reporting them to the developers is a need. This need is particularly pronounced in the case of APR techniques that are able to explore large search spaces and finding genuine patches that are reported after tens of incorrect ones (e.g., in [10]).

This paper introduces ObjSim, a lightweight, fully automatic patch prioritization tool based on object similarity heuristic. Users of test case based G&V APR techniques designed for JVM-based languages [22] are the envisioned users of this tool. ObjSim prioritizes patches that are more likely to be correct so that the users of APR techniques spend less time examining the generated patches. The tool works by comparing system state at the exit point(s) of the patched method before and after patching and prioritizing patches that result in system state that is more similar to that of original, unpatched version on passing test cases while less similar on failing

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3404362

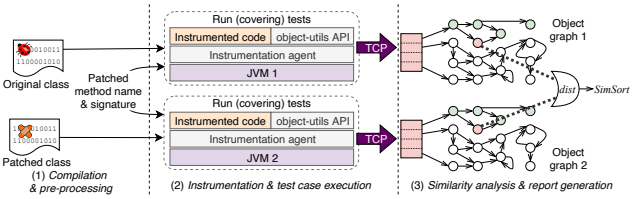

Figure 1: Overall workflow of ObjSim, given class files for original and patched classes, full name of the patched method, and the set of test cases covering the patched method. DFS traversal of object graphs and distance calculator for computing similarity of objects used in *SimSort* is also depicted.

tests. The idea is that the behavior of the original program on passing test cases can be used as a partial specification of the desired behavior of the system, so we want a plausible patch to not only not to break passing test cases but also end up in a state similar to that of original version on passing tests. Meanwhile, we want the patch to deviate from original version on failing tests, thereby requiring patches to avoid known erroneous states.

We have applied ObjSim on 358 plausible patches produced by the recent APR tool PraPR for 55 fixable bugs from Defects4J v1.4.0 [6]. We conducted our experiments on a commodity PC and set a time limit of 5 minutes and 16 GB of heap space for each bug. Compared to the original ranking scheme of PraPR, ObjSim prioritizes 5 more genuine fixes in top-1 position (16.67% improvement) and reduces average rank of genuine fixes from 3.04 to 2.74 (almost 10% improvement), yet by relying only on runtime data, it is JVM language agnostic. ObjSim, along with more details about our experiments, is publicly available [8].

2 A TOP-DOWN OVERVIEW OF OBJSIM

Virtually all available G&V APR techniques make one-point, or at most one-hunk, changes to the subject programs [7]. Thus, we built ObjSim based on the assumption that patching happens inside a single method. The basic idea of similarity-based patch prioritization is to compare system state at the exit point(s) of the patched method before and after patching and prioritizing patches that result in system state that is more similar to that of original, unpatched version on passing test cases while less similar on failing tests.

Figure 1 depicts an architectural overview of ObjSim and the steps taken for calculating similarity of system state at the exit point(s) of patched method between original and patched versions. ObjSim starts by obtaining class files for original and patched versions of the patched class and full name of the patched method. Some APR tools (e.g., [10]) already provide the needed artifacts/information; in case any of these are absent (e.g., [15] that produces only a source-level patch file), we may obtain them by applying the patch on the corresponding source file, compiling the file, and obtaining the name of the patched method via diffing.

The tool then instruments the original and patched versions of the patched method so that the instrumented program will capture a snapshot of the system state at every exit point of the patched method. This is done with the help of Java Agent technology [16] and using Javassist library [3] which allows inserting *after advices* in the form of `finally` blocks. The instrumentation code uses Java reflection to capture and serialize the object graphs reachable from all the parameters of the patched method. Then separate processes (i.e., JVM 1 and JVM 2 in Figure 1) are created to execute passing/-failing tests against the instrumented programs while containing side-effects of test execution. Each version of instrumented program is once executed against (covering) originally passing test cases and once against (covering) originally failing tests. Note that restriction to only covering test cases is not necessary and it is done solely for speeding up the entire process. Note also that the two JVM instances can run in parallel to further speed up the prioritization process. Each round of test case execution results in a number of system state snapshots for original and patched version of the instrumented program.

We use $S(I, m, t)$ to denote the set of snapshots of system state at the exit point(s) of a method m in the instrumented program I resulting from executing covering test case t. Note that $|S(I, m, t)| \geq 1$, as a method might be called multiple times. Given a patch π targeting method m in the program I, ObjSim obtains $S(I, m, t)$ and $S(\pi(I), m, t)$, where $\pi(I)$ denotes program I with patch π applied on it, for all tests t of the program. Note further that these sets are constructed inside separate processes (namely JVM 1 or JVM 2 in Figure 1), so we have to send them over to the parent process. ObjSim establishes TCP connections between itself and the child processes and transfers the recorded sets by writing them on the socket. It is worth noting that serializing arbitrary Java objects is a non-trivial engineering undertaking and we omit its details here due to space limitations. We encourage the readers to visit the website of the companion library `object-utils` for more information [9].

Except in the case of patching operations that are regarded as *anti-patterns* [18], and are usually avoided by modern APR techniques, patching a program does not change its control flow in significant ways. Thus, the number of times a method gets called tends to be the same before and after patching, i.e., $|S(I, m, t)| = |S(\pi(I), m, t)|$ for all tests t. In case the condition does not hold, ObjSim puts the corresponding patch in a bucket W, which is to be ranked based on suspiciousness values (e.g., Ochiai suspiciousness) of the patched locations. For a given program I with the set of test cases $t_1, \ldots, t_n$, a patch $\pi \notin W$ targeting method m, we use $l(\pi)$ to denote the sequence $\langle |S(\pi(I), m, t_1)|, \ldots, |S(\pi(I), m, t_n)| \rangle$.

Let P be the set of all plausible patches generated by the underlying APR tool. Let $\sim$ denote a binary relation over P defined as $\pi_1 \sim \pi_2$ iff $l(\pi_1) = l(\pi_2)$, where $\pi_1, \pi_2 \notin W$. It is easy to see that $\sim$ is an equivalence relation which is simply relating patches that have sequence of system state snapshots of the same length, both before and after patching. Let $\mathcal{Q} = P/\sim = \{[\pi]_\sim \mid \pi \in (P - W)\}$ be the quotient set of P induced by $\sim$, which is simply the set of sets of patches that have the same value for l.

Having the sets W and $\mathcal{Q}$, ObjSim produces the final ranking by concatenating m non-empty sequences $\sigma_1, \ldots, \sigma_m$ where σ_i is either $\langle \pi \rangle$ with $\pi \in W$ or $SimSort(Q)$ with $Q \in \mathcal{Q}$. $SimSort(Q)$ denotes a sequence of patches in Q that is sorted according to similarity-based criteria. Clearly, $m = |W| + |\mathcal{Q}|$. The final sequence is sorted in such a way that σ_i precedes σ_j iff $MaxSusp(\sigma_i) \geq MaxSusp(\sigma_j)$, where $MaxSusp(\sigma)$ denotes the maximum suspiciousness value (e.g., Ochiai) for the patched locations corresponding to the patches

in sequence σ. In what follows, we present a more detailed explanation on the algorithm for computing *SimSort*.

2.1 Computing *SimSort*

Given a set Q of patches, *SimSort* returns a sorted sequence of the patches in Q. Sorting is done by assigning a score to each patch in Q and putting the patches according to their scores in descending order. In order to present the algorithm in a more precise way, we define distance matrix $D = [d_{ij}]_{|Q|\times n}$, where n is the number of all test cases in the program. Each row in D corresponds to a patch in Q and each entry of the matrix represents the *average distance* of system state in the patched program from that of the original program under some test t. Specifically, $d_{ij} = \text{avg}\{dist(S_o, S_p) \mid S_o \in S(I, m, t_j) \wedge S_p \in S(\pi_i(I), m, t_j)\}$ where t_j is a failing or passing test case. The function *dist* computes distance between two objects. A more detailed description of this function is presented in the subsection that follows.

SimSort uses D to compute scores matrix $R = [r_{ij}]_{|Q|\times n}$ as follows. For each $1 \le j \le n$, the function sorts j^{th} column of D in ascending (descending) order if t_j is a passing (failing) test case. r_{ij} is the position of π_i in the sorted j^{th} column of D. The function obtains score of each patch π_i by averaging i^{th} row of R. The final result returned by *SimSort* is obtained by sorting the patches based on their scores in reverse order.

2.1.1 Computing dist. Given two objects s_1 and s_2 (that could be system states), $dist(s_1, s_2)$ is computed via DFS traversal of the object graphs reachable from s_1 and s_2 and accumulating distances of "sub-objects" of the objects in a recursive manner. Specifically, *dist* is defined recursively as follows.

- $dist(s_1, s_2) = 0$ if s_1, s_2 are equal references or equal primitive-typed objects.
- $dist(s_1, s_2) = 1$ if s_1, s_2 are unequal primitive-typed objects of the same type.
- $dist(s_1, s_2) =$ Levenshtein distance between s_1 and s_2, if s_1, s_2 are arrays of the same component type.
- $dist(s_1, s_2) = \sum_{i=1}^{n} dist(v(f_i, s_1), v(f_i, s_2))$ if s_1, s_2 are objects of the same type τ and $f_1, \ldots, f_n$ are the names of the fields declared or inherited by τ. Furthermore, $v(f, o)$ is defined to be the value of field f for object o.
- $dist(s_1, s_2) = +\infty$ if s_1, s_2 are objects of different types.

For the sake of simplicity in presentation, we have assumed that the object graphs reachable from s_1, s_2 are acyclic. Many engineering details are also omitted. Please see our implementation [9] for more details. The rationale behind the above rules is to extend Levenshtein distance algorithm [21] to handle arbitrary objects: the distance between a primitive-typed value and another of the same type is 1, the distance between arrays is computed as per the conventional Levenshtein distance algorithm, object distances are computed field-wise in a recursive manner, and the distance between objects of different types is defined to be positive infinity.

3 OBJSIM USAGE

After checking out ObjSim from [8] and installing it on the local Maven repository, the tool will be available in the form of a Maven plugin. In order to use ObjSim to prioritize plausible patches, the user needs to add the following snippet under `<plugins>` tag in the POM file of the target project and list fully qualified names of the failing test cases under the designated tag.

```
<plugin>
    <artifactId>objsim</artifactId>
    <groupId>edu.utdallas</groupId>
    <version>1.0-SNAPSHOT</version>
    <configuration>
        <failingTests>
            <!-- list of failing tests -->
        </failingTests>
    </configuration>
</plugin>
```

The tool expects a CSV file, named `input-file.csv`, under the base directory of the project. The input file is intended to contain information about the patches. Each row of this file describes a patch and has to have the following format.

```
Id,Susp,Method,Class-File,Covering-Tests
```

Where `Id` is a unique integer identifier of the patch corresponding to the line, `Susp` is the suspiciousness value for the patch location, `Method` is the fully qualified name of the patched method used during instrumentation, `Class-File` is the name of the compiled class file of patched class, and `Covering-Tests` is the space-separated list of test cases covering the patched location. Test case names should always be of the form `ClassName.MethodName` where `ClassName` is the fully qualified name of the test class. It is worth noting that we have shipped ObjSim with a tool to construct the input CSV file from fix reports generated by PraPR.

After setting up ObjSim, the tool can be invoked via the command `mvn edu.utdallas:objsim:validate`. The output of the tool shall be a sorted list of patch identifiers stored in a text file under the base directory of the target project. For more information and a demo, please see the companion video at https://bit.ly/2K8gnYV.

4 RELATED WORK

Automated patch classification has lately attracted the attention of APR research community [17, 23–26, 28]. ObjSim is most related to DiffTGen [23] and the technique introduced in [24]. DiffTGen identifies overfitted patches by finding semantic differences between the original, buggy program and its patched version by presenting values of variables and fields to the user and asking them to decide if the demonstrated behavior is reasonable. On the other hand, [24] is fully automatic and works by comparing execution traces between the original and patched programs near the patched method. Although there is a recent study using DiffTGen for classifying patches [27], it is still unclear whether or not asking the users to decide if a behavior is desired (esp., by printing out the intermediate results of computations) is cost-effective. Also, [24] records details about program execution that might be unnecessary when we reason about final results of computations; not to mention that despite discarding information from recorded traces, the implementation still calls for a large amount of computational resources. Unlike DiffTGen, ObjSim automates the process by computing the similarity between system state at the exit point(s) of the patched method in the original program and its patched version. And unlike

[24], it is lightweight yet it does not need to dismiss information about program behavior.

A body of research is also dedicated to techniques for repairing programs while minimizing semantic difference between the original and the patched versions. Chandra et al. [1] introduce a technique for identification of expressions in a buggy program that if replaced with a *good* repair candidate, will solve the bug. A good repair candidate is the one that corrects the failing executions, yet does not break passing tests. This idea forms the basis of the technique for repairing reactive programs by taking a buggy program as a partial specification of the desired behavior and producing high-quality repairs by deviating from it as least as possible [19]. This is related to Qlose [4], a technique for synthesizing fixes for small-scale student programs to pass all the test cases while the difference between execution traces of the original and the patched versions remains minimal. Although these works are related to ObjSim in the basic idea of comparing runtime state of a given patch with that of original version, the goal of two lines of research is fundamentally different. While ObjSim strives for achieving scalability in patch prioritization and applicability to a wide range of APR techniques [2, 15, 20] and programming languages, Qlose and [19] aim to synthesize a patch that is correct by construction and neither scalability nor applicability are concerns.

Pattern-based repair techniques [10, 12, 14] generate patches based on the transformation operators learned from real-world bug fix commits with the goal of generating patches that are more likely to be correct. Anti-patterns [18] refer to the transformation operators that commonly lead to plausible but incorrect patches. Similar pattern-based patch prioritization is employed by [10, 20]. ODS [26] uses source code level features to discriminate correct patches from incorrect ones. Unlike these techniques, ObjSim does not depend on program text, so it is JVM language agnostic and can be used to prioritize patches generated for programs written in languages other than Java, e.g., Kotlin or Scala.

We conclude this section by discussing other techniques. In [25], Yang et al. introduce OPad that automatically filters out overfitted patches introducing regression by generating test cases so as to fuzz test the patched method and identify patches that manifest predefined erroneous behavior (e.g., memory leak or crash). Recently, Gao et al. introduce Fix2Fit [5] that follows a similar approach. These techniques are not expected to be effective in case of programs written in managed programming languages [24]. Le et al. [13] show that semantic-based APR techniques also suffer from overfitting.

5 CONCLUSIONS

This paper presents ObjSim, a fully automatic, lightweight similarity-based patch prioritization tool for JVM-based languages. It works by comparing the system state at exit point(s) of the patched method between original program and its patched version, and prioritizing patches that result in system state that is more similar to that of original version on passing tests while less similar on failing tests. The key to scalability of ObjSim is to record and compare computed object graphs rather than complete execution traces. We observed that this technique can be quite effective, resulting in 16.67% improvement compared to default ranking scheme of state-of-the-art PraPR, yet, being semantic-based, the technique is language agnostic.

ACKNOWLEDGEMENTS

The author thanks ISSTA reviewers for their insightful comments.

REFERENCES

[1] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. 2011. Angelic debugging. In *ICSE*. 121–130.
[2] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *TSE* (2019).
[3] Shigeru Chiba. 2000. https://bit.ly/2UmMuIT. Accessed: April 2020.
[4] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program repair with quantitative objectives. In *CAV*. 383–401.
[5] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-avoiding Program Repair. In *ISSTA*. 8–18.
[6] Gregory Gay. 2017. http://bit.ly/2vxSQwR. Accessed: April 2020.
[7] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic software repair: A survey. *TSE* (2017), 34–67.
[8] Ali Ghanbari. 2020. http://bit.ly/2I3aMBU. Accessed: April 2020.
[9] Ali Ghanbari. 2020. https://bit.ly/2U4SUxt. Accessed: April 2020.
[10] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical Program Repair via Bytecode Mutation. In *ISSTA*. 19–30.
[11] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *CACM* (2019), 56–65.
[12] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *SANER*. 213–224.
[13] Xuan Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. *ESE* (2018), 3007–3033.
[14] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-Based Automated Program Repair. In *ISSTA*. 31–42.
[15] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java (Demo). In *ISSTA*. 441–444.
[16] Oracle Corporation. 2020. Java Agent. https://bit.ly/3czmzFV Accessed June, 2020.
[17] 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*. 532–543.
[18] Shin H. Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *FSE*. 727–738.
[19] Christian von Essen and Barbara Jobstmann. 2015. Program repair without regret. In *CAV*. 26–50.
[20] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *ICSE*. 1–11.
[21] Wikipedia contributors. 2020. Damerau–Levenshtein distance — Wikipedia, The Free Encyclopedia. https://bit.ly/2BrMOAj Accessed June 2020].
[22] Wikipedia contributors. 2020. List of JVM languages — Wikipedia, The Free Encyclopedia. https://bit.ly/3714hvf Accessed June, 2020.
[23] Qi Xin and Steven P Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *ISSTA*. 226–236.
[24] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *ICSE*. 789–799.
[25] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In *FSE*. 831–841.
[26] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2019. Automated Classification of Overfitting Patches with Statically Extracted Code Features. *arXiv* (2019).
[27] He Ye, Matias Martinez, and Martin Monperrus. 2019. Automated Patch Assessment for Program Repair at Scale. *arXiv* (2019).
[28] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2018. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the Nopol repair system. *ESE* (2018), 33–67.