ASAP-Repair: API-Specific Automated Program Repair Based on API Usage Graphs

Sebastian Nielebock
sebastian.nielebock@ovgu.de
Otto-von-Guericke University Magdeburg
Germany

Paul Blockhaus
paul.blockhaus@ovgu.de
Otto-von-Guericke University Magdeburg
Germany

Jacob Krüger
j.kruger@tue.nl
Eindhoven University of Technology
The Netherlands

Frank Ortmeier
frank.ortmeier@ovgu.de
Otto-von-Guericke University Magdeburg
Germany

ABSTRACT

Modern software development relies on the reuse of code via Application Programming Interfaces (APIs). Such reuse relieves developers from learning and developing established algorithms and data structures anew, enabling them to focus on their problem at hand. However, there is also the risk of misusing an API due to a lack of understanding or proper documentation. While many techniques target API misuse detection, only limited efforts have been put into automatically repairing API misuses. In this paper, we present our advances on our technique **A**PI-**S**pecific **A**utomated **P**rogram Repair (ASAP-Repair). ASAP-Repair is intended to fix API misuses based on API Usage Graphs (AUGs) by leveraging API usage templates of state-of-the-art API misuse detectors. We demonstrate that ASAP-Repair is in principle applicable on an established API misuse dataset. Moreover, we discuss next steps and challenges to evolve ASAP-Repair towards a full-fledged Automatic Program Repair (APR) technique.

CCS CONCEPTS

• **Software and its engineering** → **Error handling and recovery**; *Software libraries and repositories.*

KEYWORDS

API Misuses, Automated Program Repair, API Usage Graphs

ACM Reference Format:
Sebastian Nielebock, Paul Blockhaus, Jacob Krüger, and Frank Ortmeier. 2024. ASAP-Repair: API-Specific Automated Program Repair Based on API Usage Graphs. In *2024 ACM/IEEE International Workshop on Automated Program Repair (APR '24), April 20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3643788.3648011

1 INTRODUCTION

An Application Programming Interface (API) is the de facto standard for *client developers* to reuse algorithms and code implemented in a library or framework developed by *API developers*. An API provides a set of *API elements* (e.g., methods, fields, data structures). When using these API elements, client developers can *misuse* them, for instance, mixing up the order of method calls or calling methods with false parameters [2]. If such a misuse causes negative behavior (e.g., software crashes, performance issues), we refer to it as *API misuse*.

In the past, many techniques for detecting API misuses have been developed [2, 4, 7, 13–15]. Commonly, these techniques infer or mine likely correct *template usages* of the API (e.g., patterns or change rules) whose violations are reported as misuses. While limited, such techniques provide a profound way to specifically detect API misuses. Importantly, they *do not require dedicated tests to detect and localize misuses*, and they provide a patch via the template usage with which the misuse has been detected. A natural way to advance research is to leverage these templates for automated repair.

In this paper, we introduce the idea of an Automatic Program Repair (APR) technique for API misuses to which we refer to as **A**PI-**S**pecific **A**utomated **P**rogram Repair (ASAP-Repair). ASAP-Repair builds on a graph-based structure representing API usages named API Usage Graphs (AUGs), which has been introduced by Amann et al. [3]. Particularly, we leverage different template usages represented as AUGs (i.e., patterns and change rules) to repair API misuses. While currently limited to repairing misuses in AUGs and not in code, ASAP-Repair's major benefit compared to state-of-the-art APR techniques is that it *does not require test cases or the execution of code to localize misuses*. We demonstrate its in-principle applicability by applying ASAP-Repair on the real-world API misuse dataset MUBench [1]. Moreover, we discuss necessary steps and challenges to evaluate and to compare ASAP-Repair with state-of-the-art APR techniques. We publish ASAP-Repair and all artifacts related to this paper in a publicly available repository.[1]

2 API USAGE AND API MISUSE DETECTION

API Usage Graphs. Amann et al. [3] have developed a graph-based structure to describe API usages in Java with the goal to improve the precision of API misuse detection. For representational purposes, we introduce AUGs with a fictive fix[2] of an API misuse depicted in Listing 1 and its AUG representation before this fix (cf. Figure 1a). This fix adds a validation for the method call `B.bar()` in line 9 by checking the condition `B.isBarable()`. In case this resolves to `false`, an alternative call to `B.bar2()` is required.

AUGs are directed, labeled, acyclic multigraphs consisting of different node (i.e., action and data) and edge types (i.e., data and control flow). *Action nodes* are depicted as rectangles representing method calls (e.g., `A.foo()`) or control structures, while ellipses visualize *data nodes*, such as constants and objects (e.g., `A`). *Data flow* is represented with solid edges connecting parameters to methods

APR '24 , April 20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0577-9/24/04
https://doi.org/10.1145/3643788.3648011

[1] https://doi.org/10.5281/zenodo.10527304
[2] Please refer to our replication package for real examples.

Listing 1: Sample code for an API misuse and fix.

```
import A;
import B;
public class misuse {
    public void method() {
        System.out.println("Start run");
        A.foo();
        noise();
-        B.bar();
+        if(B.isBarable()){
+            B.bar();
+        } else{
+            B.bar2();
+        }
        A.foobar();
        System.out.println("finished run");
    }
}
```

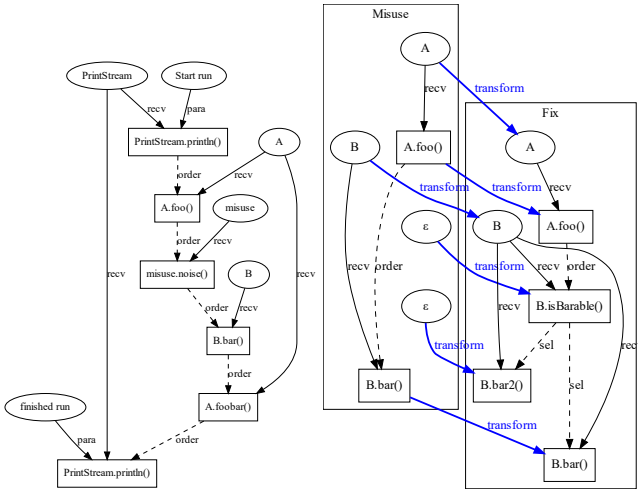
(i.e., `para`) or objects to their object methods (i.e., `recv`). *Control flow* is displayed via dashed edges that indicate the order of action nodes (i.e., `order`) or the selection after `if`-conditions (i.e., `sel`). AUGs only represent the API usage of a single method declaration (e.g., `method()`), and thus are restricted to the intra-procedural level. However, they have been demonstrated to improve API misuse detection and were enhanced as well as used by different researchers [3, 7, 13]; making them a valid basis for an APR technique.

Frequent API Usage Patterns. Researchers have proposed a set of API misuse detectors, many of them applying frequent pattern mining [2, 4, 7, 14, 15]. Their conjecture is that the way APIs are frequently used also represents their correct usages. Thus, they conduct pattern mining to infer API usage patterns and compare those with actual usages of that API. If a usage violates this pattern, it is denoted an API misuse. However, recent work showed that such misuse detectors typically suffer from a high false positive rate (i.e., a huge number of correct usages reported as misuses) [2, 5], and thus research has focused on improving the precision of misuse detectors. In the following, we assume that a pattern in the form of an AUG has been found that can correctly detect an API misuse.

API Change Rules. Another idea leverages information from already fixed misuses by extracting API code changes using version control [11, 13]. In detail, such techniques infer so-called correction or change rules, which represent the essential changes needed to fix an API misuse. A rule has the form $m \rightarrow f$ where m and f are the misuse and fixed (sub-)AUGs of the change, respectively. We show the rule for the fix in Listing 1 in Figure 1b. To match both graphs, heuristics are applied, which we also apply for ASAP-Repair (cf. Section 3). A rule consists of the subgraphs describing the misuse and its fix as well as `transform`-edges representing how nodes are transformed into their respective fix. Additions are symbolized by special ϵ-nodes indicating "holes" in the misuse graph. Similarly, deletions are represented by using ϵ-nodes in the fixed AUG.

To detect API misuses, existing techniques measure the similarity[3] sim of a candidate API usage u and both subgraphs m and f of the rule. A usage u is reported as a misuse if $sim(m, u) > sim(f, u)$ holds, meaning that the usage is more similar to the misuse part than to the fixed part. Results indicate that such a technique can achieve high precision, but suffers from a very low recall [12].

[3]Originally, this was called "distance," which is mathematically restricted. Thus, we use the more general term "similarity."



(a) AUG of the sample code. **(b) Change rule of the misuse fix.**

Figure 1: AUG (left) and change rule (right) for the example code in Listing 1.

3 PROCESS OF ASAP-REPAIR

We depict ASAP-Repair's concept in Figure 2. It fixes API misuses in the form of AUGs using *patterns* (i.e., as AUGs) or *change rules*. Both variants start with an API usage (Ⓐ), which is transformed into its respective AUG followed by the misuse detection. For the pattern-based version (Ⓑ), we can apply the violation-based technique by Amann et al. [3]. When using change rules (Ⓒ), we can use the similarity-based technique [11, 12]. If a misuse is detected, ASAP-Repair has different steps to match the nodes of the misuse AUG with those of the template AUG (i.e., pattern or change rule) through which the misuse was detected. Using this matching, we identify which nodes have to be changed (i.e., add, delete, update).

Matching Heuristic. Matching graphs refers to the subgraph isomorphism problem known to be NP-complete [6]. Thus, we adapted the strategy to produce change rules [12, 13] for graph matching. In detail, we applied the Kuhn-Munkres algorithm [10] to find a heuristic solution for the matching. This means that we create a bipartite graph based on the two AUGs (e.g., aug_A and aug_B), with one partition containing the nodes of aug_A and the other the nodes of aug_B. Then, we equalize the cardinalities of both partitions by adding special ϵ-nodes. These describe the addition and deletion of nodes (e.g., a node changed to an ϵ-node represents a deletion) within the matching. Finally, we add edges between the nodes of both partitions, which we label with the costs to transform the respective node into the other (i.e., the number of node relabelings as well as adding, deleting, and relabeling incoming and outgoing edges). The Kuhn-Munkres algorithm finds a matching that minimizes the overall costs. Note that this matching can be invalid, since it ignores the order of nodes and multiple matchings are possible.

Pattern-Based Matching. In case we repair an API misuse based on patterns, we cannot directly match the misuse with the pattern AUG. The reason is that the pattern is typically smaller (in terms of number of nodes) than the misuse. Thus, a matching would indicate that every node not part of the pattern must be deleted. We avoid

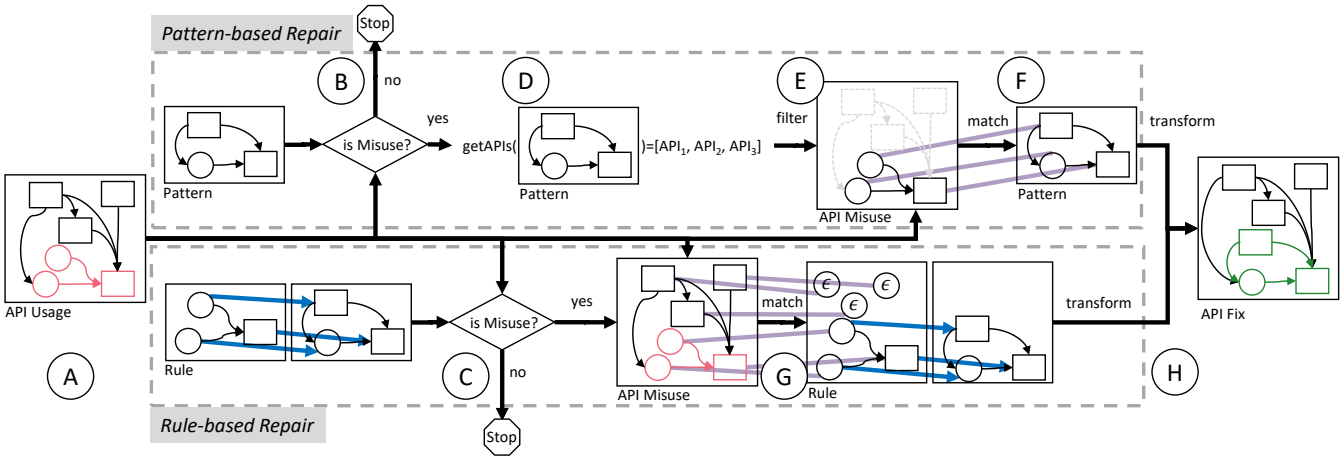


Figure 2: Concept of Pattern- and Rule-based API-specific Automated Program Repair

this issue by conducting a preprocessing of the misuse to find and match only those nodes that relate to the pattern. Particularly, we leverage that AUGs contain information on the API type of single nodes (e.g., `java.util.List`). Then, we consider only those nodes of the misuse for matching that have the same API type as the nodes in the pattern. Thus, we determine the API types present in the pattern AUG (Ⓓ) and filter nodes from the misuse AUG to extract a subgraph, consisting only of nodes of these types (Ⓔ). Then, we match only this subgraph to the pattern (Ⓕ).

Rule-Based Matching. Matching an API misuse with a change rule is simpler, since such rules describe exactly which nodes have to be changed. In detail, we match the misuse AUG with the misuse part AUG of a change rule (Ⓖ). If the rule contains an addition of nodes (i.e., the misuse part contains ϵ-nodes), we temporarily disregard these from the matching procedure. Then the matching can have three possible cases:

(1) A node from the misuse AUG is matched to a non-ϵ-node of the misuse part AUG. This indicates that this node has to be either deleted or updated (i.e., depending on whether this node is connected via a `transform`-edge to an ϵ-node or not).
(2) A node from the misuse AUG is matched to a generated ϵ-node in the misuse part (i.e., due to the cardinality equalization step of the matching). Then, this node will not be part of the transformation, since it cannot be matched to the change represented by the rule.
(3) For every disregarded ϵ-node in the misuse part AUG of the rule, we add a respective ϵ-node in the misuse AUG as well. We match those to the counterparts in the misuse part AUG to indicate that nodes have to be added according to the fixed part AUG of the change rule.

This way, we obtain a triple matching between misuse AUG, misuse part AUG, and fix part AUG. Note that nodes of the misuse AUG falling under case (2) are not part of the final repair step.

AUG-Based Repair. In the final step of ASAP-Repair, we transform the misuse graph into the fixed version (Ⓗ). We refer to the changes indicated by the respective matching, either pattern-based or rule-based, as *corrections*. We distinguish between three cases of transformations/corrections:

(1) *A non-ϵ-node from the misuse is matched to a non-ϵ-node in the correction*: Then, ASAP-Repair *updates* the node by updating its label, node type, as well as adding, deleting, updating the respective incoming and outgoing edges.
(2) *An ϵ-node from the misuse is matched to a non-ϵ-node in the correction*: ASAP-Repair *adds* the non-ϵ-node to the AUG, which also contains the addition of the respective edges represented by the correction.
(3) *A non-ϵ-node from the misuse is matched to an ϵ-node in the correction*: ASAP-Repair *removes* the non-ϵ-node from the AUG, which also contains the deletion of the respective incoming and outgoing edges of this node.

Note that ASAP-Repair keeps nodes unmatched (i.e., filtered out in the pattern-based repair or case (2) in the rule-based repair), except by changing incoming or outgoing edges from previously changed neighbor nodes. Since the Kuhn-Munkres algorithm does not always produce a valid matching, it may happen that the transformed AUG contains cycles, which results in an invalid AUG. We tackle this problem by conducting a graph cycle check. If the transformed AUG contains cycles, we retry the transformation by testing another matching, which we obtain based on an internal data structure of the Kuhn-Munkres algorithm. In case none of the possible matchings produces a valid AUG, no repaired AUG is generated.

4 PRELIMINARY RESULTS

We conducted a preliminary evaluation of ASAP-Repair by applying it to MUBench,[4] a dataset consisting of real API misuses that has been constructed by Amann et al. [1]. More precisely, we used the subset of 116 misuses provided in the replication package by Nielebock et al. [11]. In our evaluation, we conducted a sanity check on whether ASAP-Repair is applicable for real misuses. That means, for every single API misuse, we used its fix as a pattern and the changes of the fixing commit as the basis for the change rule. This way, we validated ASAP-Repair in an ideal situation, in which a perfectly matched pattern and change rule are found. Therefore, we cannot draw general conclusions about the applicability of ASAP-Repair

[4] https://github.com/stg-tud/MUBench/

Table 1: Results of repairing 116 misuses form MUBench: D - #generated data structures for repair (i.e., pattern AUG or change rule), C - #created, V - #valid, and U - #unique fixes.

	D (%)	C (%)	V (%)	U (%)
pattern-based	110 (94.8%)	61 (52.6%)	34 (29.3%)	11 (9.5%)
rule-based	86 (74.1%)	38 (32.8%)	27 (23.3%)	4 (3.4%)

in practice or in comparison to state-of-the-art APR techniques, which are subject to our future work.

For each example in the dataset, we downloaded the code version directly before and after a fixing commit, representing the misuse and fixed version, respectively, and used them to mimic the pattern and change rule. For each repair, we set a timeout of five minutes. Then, the first author checked how many repaired AUGs ASAP-Repair generated and manually validated each fix using an AUG comparison technique to decide whether the AUG of the fixed version and the generated fix were semantically equal. In Table 1, we summarize the results. We observed that ASAP-Repair produced more valid pattern-based fixes than rule-based ones (i.e., 34 vs 27) with more unique fixes (i.e., 11 vs. 4). However, the rule-based repair obtained a larger proportion of valid fixes within the generated fixes (i.e., $27/38 \approx 71\%$ vs. $34/61 \approx 55.7\%$), *indicating a better precision of the rule-based repair*. We also analyzed the reasons why the other repairs could not be generated or were invalid. For the pattern-based repair, major issues were that fixed AUGs contained cycles (36 cases), invalid edges in the fixed AUG (16 cases), or timeouts during the repair (11 cases). For the rule-based repair, we encountered timeouts and out-of-memory exceptions (59 cases) as well as invalid edges in the fixed AUGs (11 cases). Moreover, we found 16 cases in which MUBench had a false misuse description causing false or no repair at all. Three of these false descriptions caused an unsuccessful pattern-based AUG generation. For more details, we refer to our replication package.

While still limited, these results are promising to further improve and apply ASAP-Repair as a full-fledged APR-technique. It provides a framework to synthesize results from pattern-based and rule-based misuse detectors in a single APR technique. The main challenge lies in improving the matching efficiency as well as prohibiting the construction of invalid fixed AUGs.

5 CONCLUSION AND PROSPECTS

We demonstrated that AUG-based repair is possible for real API misuses. To obtain a valid APR technique, we will continue addressing the following prospects.

From AUGs to Code. While AUGs represent a good visual way to describe required API fixes, a more practical approach is a full-fledged technique producing repaired source code. We can imagine two possible ways to achieve this: (1) We directly transform an AUG into its respective source code. This requires a valid specification on how to transform code into an AUG, which is only implicitly given by the implementation[4] of MUBench [3]. Moreover, we need the back-transformation from AUGs to code. (2) We use the matching to perform the respective code transformations. In detail, we need to define for each possible node and edge transformation in the AUG a proper abstract-syntax-tree transformation.

Representative Misuse Datasets. While MUBench [1] is a valid misuse dataset, its representativeness is debatable, as many entries represent essentially the same misuse. Thus, other misuse datasets, such as AU500 [7] with its manually labeled misuses, must be used, too. However, these are limited in their size, which, in turn, limits external validity. Thus, larger validation datasets must be build.

Comparison to State-of-the-Art APR Techniques. Many APR techniques have been developed in the past [9], to which we have to compare ASAP-Repair. While we validated ASAP-Repair manually and statically, we also have to validate whether the code fixes imply a correct dynamic behavior. This requires the execution of the code using test cases. A good starting point is the APIARTy framework by Kechagia et al. [8], which analyzes state-of-the-art APR techniques on API misuses that were validated via tests. However, we noticed that some misuses are not replicable, since repositories became outdated. Thus, their build process does not work anymore. So, the underlying misuse data and build commands have to be updated manually to ensure a valid evaluation and comparison.

REFERENCES

[1] Sven Amann, Sarah Nadi, Hoan A. Nguyen, Tien N. Nguyen, and Mira Mezini. 2016. MUBench: A Benchmark for API-Misuse Detectors. In *Proc. 13th Int. Work. Min. Softw. Repos. (MSR)*. ACM, 464–467. https://doi.org/10.1145/2901739.2903506

[2] Sven Amann, Hoan A. Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. 2019. A Systematic Evaluation of Static API-Misuse Detectors. *IEEE Trans. Softw. Eng.* 45, 12 (2019), 1170–1188. https://doi.org/10.1109/TSE.2018.2827384

[3] Sven Amann, Hoan A. Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. 2019. Investigating Next Steps in Static API-Misuse Detection. In *Proc. 16th Int. Work. Min. Softw. Repos. (MSR)*. IEEE, 265–275. https://doi.org/10.1109/MSR.2019.00053

[4] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining Specifications. In *Proc. 29th Symp. Princ. Program. Lang. (POPL)*. ACM, 4–16. https://doi.org/10.1145/503272.503275

[5] Claire Le Goues and Westley Weimer. 2009. Specification Mining with Few False Positives. In *Proc. 15th Int. Conf. Tools Algorithms Constr. Anal. Syst. (TACAS)*. Springer, 292–306. https://doi.org/10.1007/978-3-642-00768-2_26

[6] Martin Grohe and Pascal Schweitzer. 2020. The Graph Isomorphism Problem. *Commun. ACM* 63, 11 (2020), 128–134. https://doi.org/10.1145/3372123

[7] Hong Jin Kang and David Lo. 2021. Active Learning of Discriminative Subgraph Patterns for API Misuse Detection. *IEEE Trans. Softw. Eng.* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3069978

[8] Maria Kechagia, Sergey Mechtaev, Federica Sarro, and Mark Harman. 2021. Evaluating Automatic Program Repair Capabilities to Repair API Misuses. *IEEE Trans. Softw. Eng.* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3067156

[9] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1, Article 17 (2018), 24 pages. https://doi.org/10.1145/3105906

[10] James Munkres. 1957. Algorithms for the Assignment and Transportation Problems. *Journal of the Society for Industrial and Applied Mathematics* 5, 1 (1957), 32–38. https://www.jstor.org/stable/2098689

[11] Sebastian Nielebock, Paul Blockhaus, Jacob Krüger, and Frank Ortmeier. 2021. An Experimental Analysis of Graph-Distance Algorithms for Comparing API Usages. In *Proc. 21st Int. Work. Conf. Sour. Code Anal. Manip (SCAM) - RENE Track*. IEEE, 214–225. https://doi.org/10.1109/SCAM52516.2021.00034

[12] Sebastian Nielebock, Paul Blockhaus, Jacob Krüger, and Frank Ortmeier. 2022. Automated Change Rule Inference for Distance-Based API Misuse Detection. arXiv:2207.06665 [cs.SE]

[13] Sebastian Nielebock, Robert Heumüller, Jacob Krüger, and Frank Ortmeier. 2020. Cooperative API Misuse Detection Using Correction Rules. In *Proc. 42nd Int. Conf. Softw. Eng. - New Ideas Emerg. Results (ICSE-NIER)*. ACM, 73–76. https://doi.org/10.1145/3377816.3381735

[14] Westley Weimer and George C. Necula. 2005. Mining Temporal Specifications for Error Detection. In *Proc. 11th Int. Conf. Tools Algorithms Constr. Anal. Syst. (TACAS)*. Springer, 461–476. https://doi.org/10.1007/978-3-540-31980-1_30

[15] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and Recommending API Usage Patterns. In *Proc. 23rd Eur. Conf. Object-Oriented Program. (ECOOP)*. Springer, 318–343. https://doi.org/10.1007/978-3-642-03013-0_15