Cooperative API Misuse Detection Using Correction Rules

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

Robert Heumüller
Otto-von-Guericke
University, Germany
robert.heumueller@ovgu.de

Jacob Krüger
Otto-von-Guericke
University, Germany
jacob.krueger@ovgu.de

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

ABSTRACT

Application Programming Interfaces (APIs) grant developers access to the functionalities of code libraries. Due to missing knowledge of how an API is correctly used, developers can unintentionally misuse APIs, and thus introduce bugs. To tackle this issue, recent techniques aim to automatically infer specifications for correct API usage and detect misuses. Unfortunately, these techniques suffer from high false-positive rates, leading to many false alarms. While we believe that existing techniques will improve in the future, in this paper, we propose to investigate a different route: We assume that a developer manually detected and fixed an API misuse relating to a third-party library. Based on the change, we can infer a *correction rule* for the API misuse. Then, we can use this correction rule to detect the same and similar API misuses in the same or other projects. This represents a cooperative technique to transfer the knowledge of API-misuse fixes to other developers. We report promising insights on an implementation and empirical evidence on the applicability of our technique based on 43 real-world API misuses.

CCS CONCEPTS

• **Software and its engineering → Error handling and recovery**; **Software defect analysis**; Collaboration in software development.

KEYWORDS

API Misuse, Misuse Detection, Bug Fix, Testing

ACM Reference Format:
Sebastian Nielebock, Robert Heumüller, Jacob Krüger, and Frank Ortmeier. 2020. Cooperative API Misuse Detection Using Correction Rules. In *New Ideas and Emerging Results (ICSE-NIER'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377816.3381735

1 INTRODUCTION

If used correctly, Application Programming Interfaces (APIs) enable developers to reuse functionalities of third-party projects or libraries. However, developers are not always familiar with the correct usage of an API, for example, mandatory orders of method-calls on a particular interface. This leads to API misuses, which can result in bugs and unexpected behavior of the software. Recently, researchers developed numerous techniques to automatically detect API misuses, for example, by inferring patterns of correct API usage from existing code and marking respective violations [1, 4, 5, 13, 17]. Still, current techniques infer high numbers of false-positive patterns, namely patterns that represent coincidental occurrences of code elements rather than correct API usages. As those "patterns" result in many false alarms during the analysis, API-misuse detection cannot be effectively applied in practice, yet.

While we believe that future improvements in API specification mining will decrease the number of false positives, we tackle the problem of misuse detection and correction from a different perspective. It is generally accepted that manual validation by skilled individuals is still the most reliable way to decide whether a pattern reflects a correct API usage [3, 16], and whether a violation of the pattern represents an actual API misuse. For example, to create the MUBench benchmark of real-world API misuses, researchers validated automatically detected misuses [3]. So, if a developer detects and corrects an API misuse, we can be rather sure that this represents a valid misuse and a correct fix. Under this assumption, if developers share their fixes with others, this can be leveraged for detecting and correcting identical misuses in other locations. However, this requires a lot of additional effort, namely, extracting the essential changes to fix an API misuse, transmitting the changes to other developers and projects, and conducting a static analysis to detect and localize corresponding API misuses.

In this paper, we propose a prototypical technique for automating these steps as well as a first empirical evaluation. In particular, we present a technique that builds on a known fix of an API misuse to infer an API *correction rule*. An API correction rule denotes a formula $A \rightarrow B$, where A represents the distilled API misuse and B its corresponding fix. Based on the applied API types and methods of the misuse part A, textual code-search engines can find similar files from different projects. Within these files, we can further analyze whether the misuse part A is contained and whether the fixing part B is not contained. Such rule violations indicate potential API misuses that are then reported to the developers. Also, part B of the correction rule can then be used to propose a solution.

2 COOPERATIVE API MISUSE DETECTION

Our notion of a cooperative API misuse detection consists of two parts. First, we need to infer the API misuse correction rule, which is a rule describing how to transform an API misuse into its respective fix. Second, we need an efficient technique to find the same misuse in other projects so that their developers can be notified.

In Figure 1, we depict a general overview consisting of the steps Ⓐ to Ⓕ. In the first step (Ⓐ), a developer manually detects an API misuse, fixes it, and commits the changes to the version control system. Since a commit may comprise several changes besides

ICSE-NIER'20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7126-1/20/05…$15.00
https://doi.org/10.1145/3377816.3381735

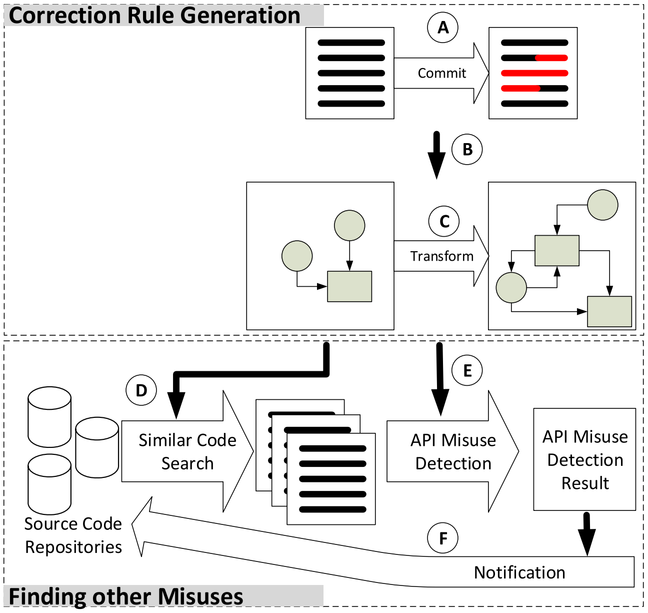

Figure 1: Concept of cooperative API misuse detection.

the actual fix, they also need to mark the method containing the misuse as well as the API(s) affected by the misuse. Based on this information, our technique automatically derives an API correction rule (Ⓑ). This rule (Ⓒ) describes how the API usage changed from the erroneous to the fixed version in terms of the relevant method's API Usage Graph (AUG). The AUG, as introduced by Amann et al. [4], is an enhanced, intra-procedural control- and data-flow-graph, which is specifically tailored for describing API usage behavior. The nodes and edges are labeled to distinguish different node types (e.g., method calls) and edge types (i.e., control vs. data flow). Based on the left-hand side of the API misuse correction rule, we first conduct a heuristic code search to identify other projects with methods that could be subject to the same misuse (Ⓓ). Next, we use the correction rule to perform a static API misuse detection on all identified methods (Ⓔ). In case of an API misuse, the developers of the respective project are notified (Ⓕ).

2.1 Inferring API Correction Rules

An API correction rule maps the subgraph of the AUG of the method with a misuse to the subgraph of the AUG of the fixed method. In contrast to existing tools, such as ChangeDistiller [7] that detects differences between two versions of an abstract syntax tree, we use AUGs as the primary data structure. We rely on AUGs, as they describe the dynamics of API usage more specifically and were successfully applied in detecting API misuses [4].

Technically, an AUG is a directed, labeled multi-graph, which can be generated for any method. Consequently, a correction rule denotes the transformation from the misuse graph into a fixed graph. However, we cannot simply use the AUGs generated from the misused and fixed method, since the AUGs would become too large and the rules would be too specific, making it impossible to compare them to other code samples. Therefore, we only include those nodes and edges from both versions of the AUG that actually changed. Particularly, we remove all nodes that were not affected by the applied fix, and filter out any changed nodes of the graphs that are not related to the misused API. This is done using the information provided by the AUG nodes, namely which API a node refers to. This way, we distill the essential part of the API misuse correction and a more general correction rule.

To obtain the essential changes, we have to identify the minimal set of edits to transform one AUG into another. This is very similar to computing the *Graph Edit Distance (GED)*. Using the GED, we measure the costs to transform one graph into another (i.e., number of added, deleted, and substituted nodes and edges) [6]. By computing the exact (i.e., minimal) GED and tracking the necessary transformations, we could therefore obtain the nodes and edges that a fix affected. However, as the computation of a GED is NP-hard [6], we use a simpler technique to determine an almost minimal set of transformations. We construct a bipartite graph, the first partition being the nodes of the misuse AUG and the second being those of the fixed AUG. Moreover, we balance the cardinalities of both partitions by introducing *empty nodes* where necessary. While the nodes of a single partition are not connected to each other, we add weighted edges from each node of the misuse partition to each node of the fix partition. The weights denote the number of transformations needed to convert a node *n* of the misuse AUG into another node *m* of the fixed AUG. We count first, whether we have to relabel node *n* with the label of *m*, and second, how many edges of *n* have a different label and/or a different source or target node (depending on whether that node is connected through an incoming or outgoing edge) compared to those of *m*. Note that an edge from an empty to a non-empty node in the bipartite graph represents the addition of a node and its respective edges, while the opposite denotes its removal. Based on this bipartite graph, we apply the Kuhn–Munkres algorithm [10] to compute a one-to-one mapping between the nodes of the partitions so that the overall costs are minimal. Thus, the result denotes the mapping that requires the least costs to transform the misuse AUG into the fixed AUG.

Finally, we generate the correction rule using the two AUGs with the added empty nodes, connecting each node from the misused AUG to the fixed AUG with a `transform` edge according to the obtained mapping. We then define a set of nodes that, together with their respective edges, will be removed from the misused and fixed AUG of that rule. This set contains (1) all nodes whose `transform` edge has a weight of zero, since these are not affected by the fix, and (2) all nodes whose label does not contain the misused API(s) denoted by the developer. In Figure 2, we depict an example of such a correction rule.

2.2 Finding Misuses in Other Projects

Based on the inferred API correction rule, we aim to find the same misuse in other locations, primarily in other projects. Technically, this requires generating the AUG for each method. Then, we check whether the sub-AUG on the left-hand side of the rule (i.e., the misuse) is a subgraph of that generated AUG, and whether the sub-AUG on the right-hand side of the rule (i.e., the corresponding fix) is not part of the generated AUG. However, the subgraph isomorphism problem is NP-complete, and therefore we can only practically apply this technique to a restricted number of AUGs.

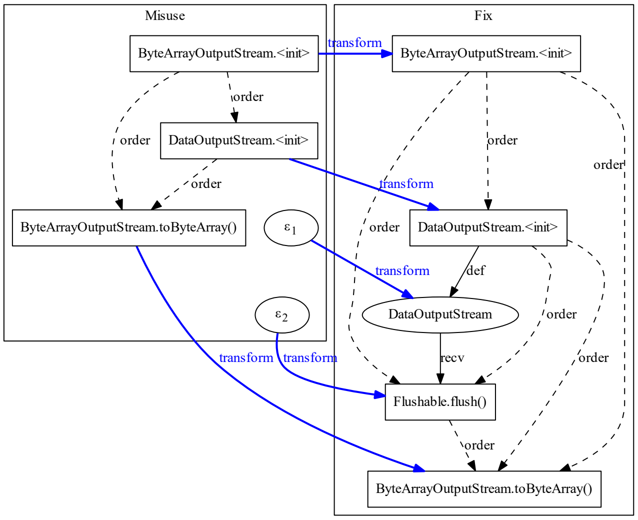

Figure 2: Example of an API misuse correction rule of a *wrapped stream misuse*. Rectangles are action nodes (e.g., method calls), ellipses are data nodes (e.g., objects), dashed black edges are control flows (e.g., order of nodes), solid black edges are data flows (e.g., defining an object or receiving an object to call a method on it), bold blue edges are node transformations, and ϵ_i represents empty nodes.

For this reason, we suggest to first filter whether the source code uses the affected API and its corresponding, misused methods. This information is provided by the API misuse correction rule and by the reported, misused API. As a lightweight technique, we can adopt textual code-search engines, such as SEARCHCODE.[1] Particularly, we can search for the misused APIs, for example, via their import statements, and for misused method calls using the labels in the misuse sub-AUGs. The result is a condensed set of files and methods that may contain the misuse. These methods are then analyzed using the subgraph isomorphism check described above, or with a static API misuse detector, for example, as introduced by Amann [4]. Finally, in case of a detected misuse, the developers will be notified, for example, via a ticket in the issue tracking system. Note that this ticket can also contain the possible fix of the misuse based on the API correction rule.

2.3 Cooperative API Misuse Detection

Ensuring that APIs are used correctly has apparent benefits, namely avoiding bugs and failures, and can also ensure user satisfaction [9]. Consequently, numerous techniques have been proposed to tackle the issue of identifying and fixing API misuses [1, 4, 5, 13, 17], but, to the best of our knowledge, none employs our idea of correction rules. In the context of cooperative and especially fork-based software-reuse, related work aims to automatically identify bug fixes [15] and to improve the propagation of updates [11, 12]. Still, these techniques usually update linked projects (i.e., forks) that have a high similarity, and thus provide no cross-project support.

We envision our cooperative API misuse detection technique to be employed as a service in online source-code repositories. Software projects that register for such a service could submit their own API correction rules to a database, and in turn profit from rules originating from other projects. In an advanced version, we could directly generate pull requests based on the correction rules.

3 APPLICABILITY STUDY

For a first evaluation, we implemented our conceptual correction-rule generator as a prototype[2] in Java. This allows us to build on an existing AUG implementation [4], and to use the misuses in the MUBench database [2] for our evaluation. Then, we analyzed (1) whether our technique can generate valid correction rules and (2) to what extent the correction rules detect API misuses.

Correction Rule Generation. For our evaluation, we used 43 of 103 (as of February 2019) real-world misuses from the MUBench benchmark [2]. We only considered misuses that involved non-standard Java APIs, as otherwise, we could hardly filter code regarding the misused API. Further, we omitted misuses that were not accessible via a version control system, ensuring that we could access an erroneous and a corrected version. Moreover, we did not consider artificially generated misuses in the benchmark and excluded 36 misuses from the JODATIME project that represented code duplicates of already included misuses. During our analysis, we noticed that in two cases the actual misuses were not fixed in the denoted commits, making it impossible to infer a correction rule. Therefore, we only considered 43 misuses for our evaluation.

After generating the correction rules, two analysts independently checked the validity of the rules, namely whether the AUG transformation reflected the misuse and fix description MUBench provides for the bug. If the two analysts disagreed, they discussed and corrected their validation accordingly. Both analysts agreed that 20 rules were valid, leaving 23 invalid correction rules. Sixteen of those rules were rejected because the AUGs did not contain the API type, and therefore the type filtering removed the nodes relating to the misuse. The reason for this lies in the static type resolution that we currently apply, which cannot derive types outside the scope of the method (e.g., a global variable declaration). So, an improvement of static type resolution could greatly benefit the rule generation. In the other cases, the internal properties of the AUG or its generation-process caused that the rule was only partially inferred or not at all. For example, in one case the fix of a misuse was the substitution of a `String`-value. Since AUGs do not contain information on actual values, this type of fix could not yet be inferred.

API Misuse Detection using Correction Rules. We aim to decrease the high false-positive rates of current API misuse detection techniques. To evaluate this goal, we checked the ability of our technique to detect misuses and to not raise false alarms for already fixed versions. As a first analysis, we selected nine misuses that represent the largest set of the same misuse in our data. This way, we can cross-check whether a rule inferred from one fix can be applied to other samples of the same bug. In particular, the misuse describes a `DataOutputStream` wrapping of an underlying `ByteArrayOutputStream`. However, the `DataOutputStream` is neither closed nor

[1] https://searchcode.com

[2] https://bitbucket.org/SNielebock/icse-2020-nier-cooperative-api-misuse

flushed before calling the method `toByteArray()` on the `ByteArrayOutputStream`. We depict the automatically inferred correction rule for this misuse in Figure 2.

For our evaluation, we used six valid rules (pairs from three projects) of our previous analysis. We then cross-checked for each rule whether it was able to detect the same misuse for one of the other eight occurrences of that misuse. Analogously, we checked whether the rule did not find a misuse in the corresponding fixed versions of those eight misuses. If the rule was not violated by the misused code, we denoted this as a *false-negative.* In case that a fixed method violated the correction rule, we classified it as *false-positive.* A violation of a correction rule denotes that the misused AUG is a subgraph of the method AUG and the corresponding fix AUG is not a subgraph of the method AUG. For simplicity, we applied a naïve approach to this subgraph isomorphism problem by checking whether the node and edge set of the sub-AUG was a subset of the node and edge set of the method AUG, respectively.

Since the considered misuse in our evaluation represents a missing method call of `flush` or `close`, the misused AUG is a subgraph of the fixed AUG, namely $A' \subset B'$, with A' and B' being the set of nodes and edges of the AUGs A and B, respectively. In case B is falsely detected as subgraph, this can be caused by the nodes and edges of (1) $A' \cap B'$, (2) $B' - A'$ or both (3) $A' \cap B'$ and $B' - A'$. For cases (1) and (3), the false subgraph detection also occurs for A. Since then neither A nor B are actually subgraphs, our technique made no error. For case (2), our technique would falsely classify a real misuse as no misuse, and thus deduce a false-negative result. So, this simplified analysis has only a negative effect on the recall.

Using this simple misuse detection technique, we determined the number of false-negatives and false-positives, based on which we computed the respective precision and recall for each correction rule. Note that we excluded the result where we checked the correction rule on its original misuse, we only checked it against other misuses. The precision ranges from 57 % to 80 %, while the recall is between 25 % and 50 %. Due to the improvable results, we further analyzed the reasons for false-negatives and false-positives. In all cases of false-negatives (i.e., non-detected misuses), the misuse sub-AUG contained additional nodes that were not part of the misuse or described the misuse in a different shape. The number of false-positives (i.e., false alarms) usually resulted from an alternative fix. For example, for the wrapped stream misuse, three of the misuses used the `close()` instead of the `flush()` method to fix the misuse. However, we could not generate such a rule, due to the problems of the static type resolution discussed before.

4 CONCLUSION

In this paper, we introduced the idea of cooperative API misuse detection based on correction rules. Our technique involves the developer by reusing their API misuse fix with the respective method and the misused API(s) to automatically infer a correction rule. We discussed how heuristic code search can find similar source files which may contain the same misuse and how actual API misuses in those files can be identified using the correction rule.

We prototypically implemented our technique and conducted a first evaluation of the validity of inferred correction rules and their ability to decrease the false-positive rate. Our preliminary results indicate that the static type resolution we used is not optimal. Still, assuming that the submitting developer has access to the complete code, it is trivial to obtain the types and integrate them in the AUGs.

A major reason for false-positives is that our rules fail to provide alternative solutions for a misuse. So, it is reasonable to cluster similar solutions and provide them as alternatives, for example, in the form of $A \rightarrow B_1 \vee B_2$. Then, a violation denotes that the AUG contains A and neither B_1 nor B_2.

Moreover, our misuse detection needs a different misuse criterion (i.e., only containing the misuse graph A) in case the fix graph B is a subgraph of the misuse graph A (i.e., $A \supset B$). Instead, false-negative rules were caused by too specific (i.e., too large) misuse graphs. By synthesizing misuse graphs from many samples, we may distill a more generic pattern. Similar to the MAPO miner [17], we plan to use clustering with subsequent pattern mining for this purpose.

We did not analyze the code search based on inferred correction rules, yet. In future work, we plan to compare such searches with alternatives, such as code clones [14] and example recommendation [8]. Finally, our technique heavily relies on the integrity of the community to submit valid fixes. However, to deal with false or malicious fixes, we plan to mine rules to remove bad examples.

ACKNOWLEDGMENTS

This research has been supported by the German Research Council DFG (grant no. SA 465/49-3).

REFERENCES

[1] Miltiadis Allamanis and Charles Sutton. 2014. Mining Idioms from Source Code. In *FSE*. ACM.

[2] Sven Amann, Sarah Nadi, Hoan A. Nguyen, Tien N. Nguyen, and Mira Mezini. 2016. MUBench: A Benchmark for API-Misuse Detectors. In *MSR*. ACM.

[3] Sven Amann, Hoan A. Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. 2018. A Systematic Evaluation of Static API-Misuse Detectors. *IEEE Transactions on Software Engineering* (2018).

[4] Sven Amann, Hoan A. Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. 2019. Investigating Next Steps in Static API-Misuse Detection. In *MSR*. IEEE.

[5] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining Specifications. In *POPL*. ACM.

[6] David B. Blumenthal and Johann Gamper. 2018. On the Exact Computation of the Graph Edit Distance. *Pattern Recognition Letters* (2018).

[7] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. 2007. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering* 33, 11 (2007).

[8] Reid Holmes, Robert J. Walker, and Gail C. Murphy. 2005. Strathcona Example Recommendation Tool. In *ESEC/FSE*. ACM.

[9] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2013. API Change and Fault Proneness: A Threat to the Success of Android Apps. In *ESEC/FSE*. ACM.

[10] James Munkres. 1957. Algorithms for the Assignment and Transportation Problems. *Journal of the Society for Industrial and Applied Mathematics* 5, 1 (1957).

[11] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. 2016. Synchronizing Software Variants with VariantSync. In *SPLC*. ACM.

[12] Luyao Ren. 2019. Automated Patch Porting Across Forked Projects. In *ESEC/FSE*. ACM.

[13] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2013. Automated API Property Inference Techniques. *IEEE Transactions on Software Engineering* 39, 5 (2013).

[14] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming* 74, 7 (2009).

[15] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying Linux Bug Fixing Patches. In *ICSE*. IEEE.

[16] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are Code Examples on an Online Q&A Forum Reliable?: A Study of API Misuse on Stack Overflow. In *ICSE*. ACM.

[17] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and Recommending API Usage Patterns. In *ECOOP*. Springer.