SCIENCE CHINA
Information Sciences

• LETTER •

May 2018, Vol. 61 050108:1–050108:3
https://doi.org/10.1007/s11432-017-9416-1

Special Focus on Internetware: Research Progress and Current Trend

Mining API usage change rules for software framework evolution

Ping YU[1,2*], Fei YANG[1,2], Chun CAO[1,2], Hao HU[1,2] & Xiaoxing MA[1,2]

[1]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing* 210023*, China;*
[2]*Department of Computer Science and Technology, Nanjing University, Nanjing* 210023*, China*

Received 29 November 2017/Revised 4 February 2018/Accepted 8 April 2018/Published online 18 April 2018

Citation Yu P, Yang F, Cao C, et al. Mining API usage change rules for software framework evolution. Sci China Inf Sci, 2018, 61(5): 050108, https://doi.org/10.1007/s11432-017-9416-1

Dear editor,
Software frameworks or libraries are frequently evolving APIs (application programming interfaces) to include new features, improve performance or fix bugs. Programs relying on these APIs should be updated according to these changes. Some modern IDEs (integrated development environments) such as Eclipse and IntelliJ IDEA are able to provide compile error reports or warnings to nonexistent or deprecated APIs. However, developers still have to search for precise replacement APIs and check their usage to fix bugs or remove warnings. To facilitate this process, many approaches have been proposed to generate API usage change rules automatically by mining software repositories [1–4]. Text similarity calculation is useful to pick out replacement methods that have enough similar signatures. Call dependency analysis is another popular approach by comparing API invocation changes. Some researchers proposed hybrid approaches that mix these technologies together, e.g., AURA [1] and HiMa [2]. We propose a context-sensitive approach named AUC-Miner (API usage change miner) to automatically extracting fine-grained invocation changes based on call dependency analysis. Our contributions are as follows. (1) Call dependency relation is refined by context of where and when a method is invoked. Specifically, context refers to the location where an API is invoked by a caller method. Different location may indicate different API replacement relationship. API invocation sequence chain is another kind of context helping to recognize correct API usage changes. (2) We adopt frequent itemset mining to generate one-to-one, one-to-many and many-to-one API usage change rules. Methods' comments are utilized for root methods that are not called in the framework itself. Different from external text documents or information from version control system, comments are part of source code and reflect design intention. We further take another process to search for root method replacements by including some method pairs that have high similarity. (3) Based on AUC-Miner algorithm, we have developed an Eclipse plug-in named AUC-Rec to assist application programmers to update their code according to framework evolution.

AUC-Miner consists of three main processes (marked as (1), (2), (3) in Figure 1). We detail the processes as follows.

(1) Mining API replacement rules. Our approach is based on call dependency analysis. Assume that in the old release there is a method m1 which invokes method m2, but in the new release, method m1 does not call m2 but adds invocation to m3. Here, we regard 'remove m2' and 'add m3' as items, and group them together to get a transaction for the caller m1 which is regarded as a seed method. Each transaction contains a set of items.

* Corresponding author (email: yuping@nju.edu.cn)

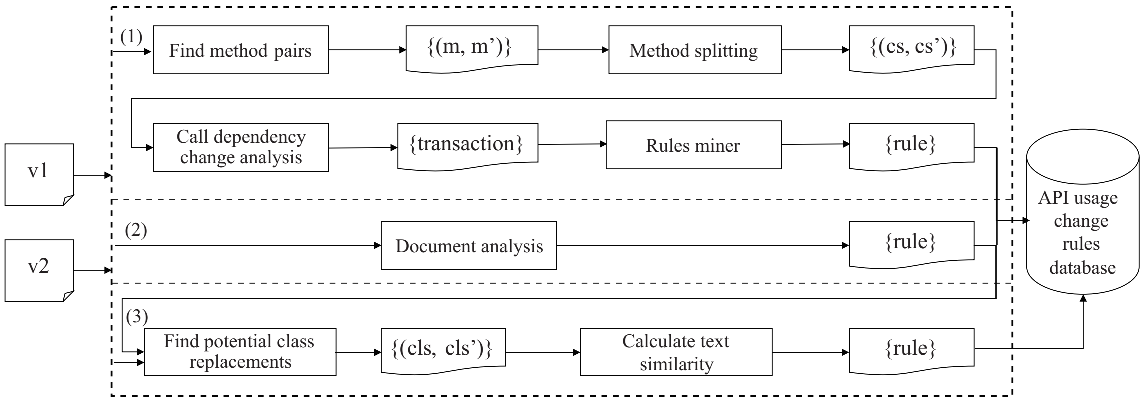

Figure 1 Overview of AUC-Miner.

First, we generate seed method pairs {(m, m')} from source codes of framework release v1 and v2. In regard to a chunk of method body, the context of the first statement is quite different from that of the last one. To extract precise transactions we need to distinguish those locations. We propose a splitting algorithm to get code snippet pairs {(cs, cs')}. Here LCS (longest common subsequence) algorithm is used to search for those common code snippets that have at least L lines of code existing both in source code of the old method and the new one. And then we use those common code snippets as separator lines to split method body into some code snippets and match snippet pairs between two versions. We further analyze call dependency changes between each code snippet pair and generate transactions {transaction} from the analysis.

Another problem is whether a method is called directly or indirectly by the seed method. Existing approaches only paid attention to the methods directly called by a target caller. However, if method invocation chain is complex, there may be some spurious replacements. For example, method m1 calls method m2 and method m2 calls m3 in the old release. In the new release m1 does not invoke m2, but invokes m3. If we just analyze changes between two sets of methods directly called by m1 in the old and new releases, we would generate an item of 'add m3'. However, this item is a spurious addition because in the old release m3 is invoked by m1 indirectly through m2. Spurious removal also exists. For example, method m1 calls m3 in the old release but in the new release m1 removes invocation to m3 while adding an invocation to m2, and m2 calls m3. If following the basic approach, an item of 'remove m3' will be generated. However, this removal item is also spurious because m3 is now called by m1 indirectly through m2.

These spurious items come from the ignorance of context about when a method is invoked. The invocation chain of a seed method is important to refine API usage changes. For this reason, we not only consider methods directly invoked but also consider methods indirectly invoked. The length of invocation chain is assigned to 3 because if the chain is too long, it would become a big graph if a method invokes many APIs.

When the transaction set is ready, we implement Apriori algorithm to mine frequent itemsets. For each frequent itemset, we obtain an association rule by taking all 'remove' items as antecedent and all 'add' items as consequent. For each candidate replacement rule $a \Rightarrow a'$, which means that methods in set a can be replaced by methods in set a', we use the following formula (1) to calculate confidence of it. In this formula, csp represents a code snippet pair. Thus $O(\text{csp})$ and $N(\text{csp})$ represent old version and new version respectively. And we use $\rightarrow$ to represent invocation relationship. Default confidence is set to 1.0.

$$\begin{aligned}&\text{conf}(a, a') \\ &\quad = \frac{\|\{\text{csp}|O(\text{csp}) \rightarrow a \wedge N(\text{csp}) \rightarrow a'\}\|}{\|\{\text{csp}|O(\text{csp}) \rightarrow a\}\|}.\end{aligned} \tag{1}$$

(2) Document analysis. In the second step, source code comments are used to extract replacement rules about root methods that are not identified in the first step. Some annotations such as @deprecated help us quickly find useful comments. Since comments are written in natural language, we use NLP technology to analyze phrases with the patterns of 'replaced by', 'use instead' or 'renamed to'.

(3) Text similarity. The third process is an iterative one. We infer potential class replacement rules based on API replacement rules generated by previous two steps. We regard two classes that are involved in the same API replacement rule as a potential class replacement pair {(cls, cls')} . For example, after getting the method replacement rule

cls.m ⇒ cls'.m' (supposing cls and cls' are class names), we recognize that class pair (cls, cls') has potential replacement relationship. Note that two classes in a pair may be the same. Second, for each potential class replacement pair (cls, cls'), we calculate text similarity of signatures between each method removed from cls and each method added to cls' to generate method replacement rules.

As Nguyen et al. [3] did, AUC-Miner calculated text similarity between methods' signatures. We use Levenshtein distance (LD) to measure the similarity of two strings. Then, the similarity between two methods is computed by weighted sum of textual similarities of return types, method names and a list of parameter types. The similarity of method names has the highest weight of 0.5, and the others are assigned the weight of 0.25.

For each replacement pair of classes, we sort all possible method replacement pairs according to similarity in descending order. And then we repeat the following process until all possible method pairs are handled: peek the method pair with highest similarity as a replacement rule and remove all remaining method pairs containing either method of the pair.

As shown in Figure 1, all API usage change rules are stored in a database. Namely AUC-Miner is also a hybrid approach as AURA and HiMa, but it focuses on method invocation context and rule mining. Based on AUC-Miner algorithm, we have implemented an Eclipse plug-in named AUC-Rec. It is designed as an online system for API replacement recommendation. It depends on the rules generated in AUC-Miner and then recommends replacement APIs for application developers. In fact, developers expect to be told how to organize their refactoring code, especially when encountering complex evolution rules. Thus, AUC-Rec provides some code examples for users to improve their refactoring efficiency. As an open source project, it can be checked out from Github[1)] and installed into Eclipse IDE (3.6 and above versions). AUC-Rec provides sub-second responsiveness for a majority of test conditions.

To evaluate AUC-Miner, we select three different sized frameworks (i.e., jEdit, Struts and Android SDK) with eight releases as experimental subjects and compare our result with basic call dependency analysis and AURA [1]. To reduce bias or mistakes, we check all rules carefully by two authors and two non-authors. In all cases, AUC-Miner generates more rules and gets averagely 18.2% higher precision than the basic approach. AUC-Miner also achieves averagely 8.6% higher precision than AURA, and generates more one-to-many and many-to-one rules than AURA. However, AURA gets more one-to-one rules than AUC-Miner. This is because AURA computes similarity among all possible methods, but AUC-Miner only calculates those root methods relevant to previously identified class pairs that have potential replacement relationship. We think the method pairs that are inferred from explicit or implicit call dependency analysis are more credible. To verify this guess, next we will do experiments on a well-maintained framework developed by our research group.

Conclusion. We propose a fine-grained approach to automatically generate API usage change rules from and only from source code of subject frameworks. In contrast to previous works, we take into account method invocation context of where and when a method is called. The root method problem is alleviated by analyzing code comments with NLP and calculating similarity of most relevant methods through weighted summation. We now only make use of framework code, but client application code and unit test code can also be utilized to enrich the transaction set. In future study, we will introduce these codes as input and design a more suitable approach to model invocation context including control structures, data dependencies and other factors.

Acknowledgements This work was supported in part by National Key R&D Program of China (Grant No. 2017YFB1001801), National Basic Research Program of China (973 Program) (Grant No. 2015CB352202), National Natural Science Foundation of China (Grant Nos. 61690204, 61472177), and Collaborative Innovation Center of Novel Software Technology and Industrialization.

References

1 Wu W, Guéhéneuc Y, Antoniol G, et al. AURA: a hybrid approach to identify framework evolution. In: Proceedings of the 32nd International Conference on Software Engineering, Cape Town, 2010
2 Meng S C, Wang X Y, Zhang L, et al. A history-based matching approach to identification of framework evolution. In: Proceedings of the 34th International Conference on Software Engineering, Zurich, 2012
3 Nguyen H A, Nguyen T T, Wilson G, et al. A graph-based approach to API usage adaptation. ACM SIGPLAN Notice, 2010, 45: 302–321
4 Kim M, Notkin D, Grossman D, et al. Identifying and summarizing systematic code changes via rule inference. IEEE Trans Softw Eng, 2013, 39: 45–62

1) https://github.com/yangfeit19/auc-miner.