Who plays with Whom? ... and How?

Mining API Interaction Patterns from Source Code

Robert Heumüller
OvGU Magdeburg
Germany
robert.heumueller@ovgu.de

Sebastian Nielebock
OvGU Magdeburg
Germany
sebastian.nielebock@ovgu.de

Frank Ortmeier
OvGU Magdeburg
Germany
frank.ortmeier@ovgu.de

ABSTRACT

State-of-science automated software engineering techniques increasingly rely on specification mining to provide API usage patterns for numerous applications, e.g. context sensitive code-completion, bug-detection or bug-fixing techniques. While some existing approaches already yield good results with respect to diverse tasks, the focus has always been on the inference of high-quality, reusable specifications for single APIs. However, in contemporary software development it is commonplace to combine a multitude of different libraries in order to increase efficiency by avoiding the reimplementation of the wheel. In contrast to prior research, in this idea paper we propose to explicitly study the patterns of interaction between multiple different APIs. First, we introduce a method for mining API interactions patterns from existing applications. Then, we give an overview of our preliminary investigation, in which we applied the method to a case-study of nearly 500 Android applications. The exemplary results show that there definitely exist valuable interaction patterns which can be helpful for various traditional and automated software engineering tasks.

CCS CONCEPTS

• **Software and its engineering** → **Software libraries and repositories**; **Reusability**;

KEYWORDS

api interaction, library interaction, specification mining

ACM Reference Format:
Robert Heumüller, Sebastian Nielebock, and Frank Ortmeier. 2018. Who plays with Whom? ... and How?: Mining API Interaction Patterns from Source Code. In *Proceedings of the 7th International Workshop on Mining Software Repositories (SoftwareMining '18), September 4, 2018, Montpellier, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3242887.3242888

1 INTRODUCTION

API specification mining[1] has attracted much attention in research throughout the last decade. In general, these methods aim at inferring different kinds of properties describing how APIs may correctly and safely be used. The largest number of techniques deals with inferring temporal API properties, i.e. constraints on the valid order of the method invocations of an API [10]. The major problem of automated specification inference algorithms has always been the large number of false-positives, i.e. extracted patterns which neither reflect a verifiable property of the API nor provide any other obvious value. Beside many types of heuristics, one basic means of reducing the otherwise prohibitively high false-positive rates is the restriction of the mining process to only a single API of interest. Therefore, only little attention was paid to patterns originating from multiple APIs [12, 13]. In contrast, in this paper we propose a method to systematically discover and study *interaction patterns* which describe recurring patterns for using two or more APIs to achieve a desired effect. It is important to note that interaction patterns can describe not only *which* API libraries or frameworks are usually applied together, but also *how* they interact, e.g. *which methods in which sequence* make up the interactions.

```
void writeMsg(OutputStream os, Message m) {
  BufferedSink bs = Okio.buffer(Okio.sink(os));
  m.encode(bs);
  bs.emit();
}
```

Listing 1: Example Interaction Pattern between `okio.BufferedSink` and `wire.Message`

Listing 1 shows an exemplary interaction pattern from the Android ecosystem. This example, which was taken from our preliminary evaluation described in Section 3, showcases a common interaction between the widespread Android APIs *okio*, a popular IO library, and *wire*, an Android-tailored implementation of Google's *protocol buffers* serialization concept. First, an `okio.BufferedSink` is allocated to decorate the `OutputStream` with an additional buffer. Next, the protobuf `Message` is serialized into the sink using `encode`, before finally flushing the buffer using `emit`.

While patterns such as the one in Listing 1 can be found by standard specification mining algorithms, this is not possible for approaches which consider only sets of preselected APIs. In general, API usage patterns have been successfully applied to a number of applications. However, we believe that interaction patterns can provide additional value in particular for the following use-cases and thus warrant a separate consideration.

Augmenting Documentation. While many interaction patterns likely exist as tacit knowledge in the respective communities, documentation is necessarily sparse due to the vast variety of possible

SoftwareMining '18, September 4, 2018, Montpellier, France
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5975-7/18/09…$15.00
https://doi.org/10.1145/3242887.3242888

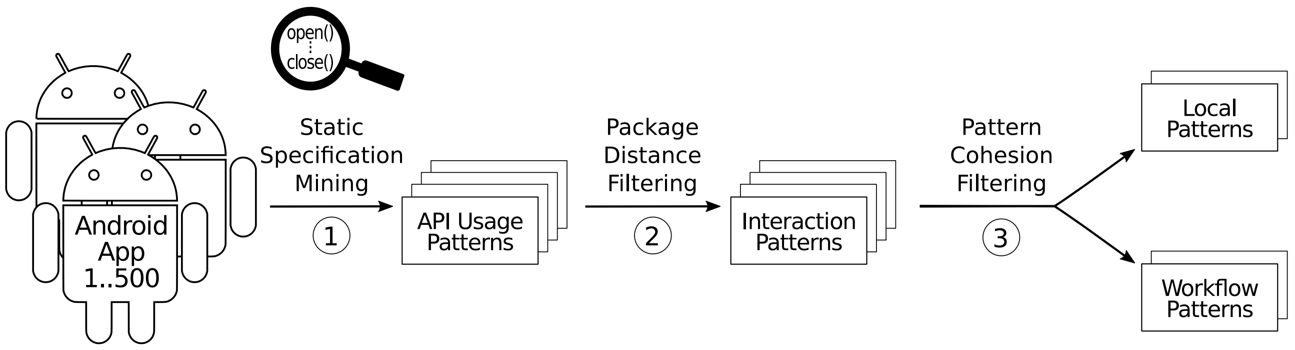

Figure 1: Process for Mining Interaction Patterns from Android Applications

combinations between APIs. Mining interaction patterns can be used to uncover new patterns or to formalize existing knowledge.

Source Code Completion. State of science engines leverage usage patterns to provide superior, context-sensitive proposals. Whereas patterns referring to single APIs are particularly effective in preventing API misuses, interaction patterns can take into account further elements of context and could thus provide more accurate recommendations.

Enhanced Library Recommendation. is possible when only a subset of the APIs of an interaction pattern is present at a recommendation site. In this case, the missing APIs can be interpreted as a recommendation for additional APIs which are likely to work well with the existing libraries. Compared to recommendation engines which only take into account *which libraries* are used in the recommendation context, using interaction patterns would enable recommendations which also take into account *how the libraries are used.*

For these reasons, we propose to systematically study interaction patterns with the goal of leveraging them to improve on the above use-cases. To avoid confusion regarding the terminology, for the scope of this paper we establish the following definitions:

Definition 1. ***(API) Events*** *refer to all atomic interactions with the interface of a library or other resource. The type of event varies depending on the resource which is exposed through the API. For example, for software libraries events are usually method-calls, whereas for web-services events may refer to the invocation of http endpoints.*

Definition 2. ***Execution Traces*** *traces consist of recorded sequences of API events. These can be either generated dynamically or by means of static analysis. Dynamic methods have the advantage of producing only traces which are feasible in practice, but require meaningful input data to provide sufficient coverage of the software to analyze. On the other hand, static approaches, which only require the program itself as input, are only capable of providing abstractions of the actual program semantics, typically yielding a number of traces which are not feasible in practice.*

Definition 3. ***(API) Usage Pattern.*** *We use this term to refer to frequently appearing, closed subsequences of API events in execution traces. In terms of semantics, we regard usage patterns as* partial-correctness temporal safety properties *[6]. Three criteria need to be met: (1) a minimum support threshold, (2) patterns must appear in at least two different methods, because static trace generation may produce large numbers of traces per individual method, and (3) we require the existence of a control- or data-flow dependency between the events of a usage pattern in order to reduce the number of false-positives.*

Definition 4. ***(API) Interaction Patterns*** *are a subset of usage patterns. In addition to the criteria above, interaction patterns must (1) exhibit events belonging to at least two different APIs and (2) must appear in at least two different projects. We further distinguish between* local patterns *and* workflow patterns. *The former depict interaction between different APIs in a close context, typically within the same method. The latter take place on an application level, e.g. between data-acquisition and data-analytics libraries. Therefore we expect their events to be spread further across the application.*

In the following sections, we first introduce our method for mining interaction patterns (Section 2), give a brief overview of our preliminary investigation - including further example patterns - (Section 3), and summarize the relevant related work (Section 4) before concluding the paper in Section 5.

2 EXTRACTING INTERACTION PATTERNS

Figure 1 gives an overview of our envisioned three-step process for extracting interaction patterns. We intend to perform our analysis of interaction patterns on a case study consisting of 500 Android applications listed on *www.androiddrawer.com.* Android apps are particularly suitable for automated analysis due to their generally well-formed structure. The large number of specimens from different categories should help to provide a representative overview of the Android ecosystem.

Step ① is to apply a (fairly-)standard specification mining process to extract API usage patterns from the case-study of Android applications. Here, we rely on static analysis in order to generate large numbers of execution traces for all applications. If we were only interested in finding *local* interaction patterns, it would suffice to use *intra-procedural* trace generation. However, to also be able to find the anticipated *workflow* patterns, an inter-procedural method is necessary, e.g. as seen in [7]. The generated execution traces are then mined for closed frequent subsequences using an off-the-shelf data-mining tool such as *spmf* [4]. The resulting candidate usage patterns are further filtered according to Definition 3.

Step ② filters out all usage patterns which involve only a single API, yielding the subset of interaction patterns. However, it is not always obvious whether or not two events should be considered as belonging to the same API or to different APIs. For example, method calls on separate but strongly related classes of the same library could be interpreted as belonging to different APIs, those of the classes, or to the unified API of the library. To avoid this binary decision, we propose a continuous scale based on a distance measure between package qualifiers. I.e., the smaller the number of shared ancestral packages between two classes appearing in a usage pattern, the larger the probability that the pattern is an interaction pattern. A threshold can then easily be selected and adjusted for the purpose of experimentation.

Step ③ categorizes each interaction pattern either as a local pattern or as a workflow pattern based on a cohesion measure: We retrieving the positions (integer indices) of where a particular pattern's events appear in an execution trace. Then, we compute an average variance of these positions across all traces which exhibit the pattern. Low average variances show high cohesion and thus indicate local patterns; for high variances vice-versa.

3 PRELIMINARY RESULTS

In order to gain a first insight into the existence of interaction patterns our envisioned Android case-study, we performed a simple preliminary investigation before implementing the process introduced in Section 2. First, we extracted lists of imported packages from 493 case-study apps. Then, we used *spmf* [4] to mine frequently recurring itemsets of up to three elements across the filtered lists using a minimum support value of 5%. The resulting sets of packages give an indication of the between which libraries to expect interaction patterns. We hand-selected some promising library pairings which appeared in 5 (SlidingMenu ex.), 29 (Protobuf ex.) and 45 (Glide ex.) of the case-study apps. Next, we used them to perform a manual, directed search for interaction patterns, both in the case-study apps and in online source-code repositories.

Beside the introductory pattern in Listing 1, we selected two additional interaction patterns, which are explained in the following.

Listing 2 depicts a local interaction pattern between two third-party libraries, the *okhttp* http client library and the *glide* image loading library. Here, a custom `GlideModule` is configured for an application in order to make the library use an `OkHttpClient` whenever *glide* is requested to load media from url resources. To achieve this, a corresponding `OkHttpUrlLoader.Factory` must be created for the client, before registering it with the `GlideModule` for the desired `GlideUrl` resource type.

In contrast, the interaction pattern in Listing 3 also shows a local interaction, but between a standard library and a third-party library. In this example, the popular `SlidingMenu` library is used to give standard `android.Activity` instances the additional functionality of sliding in and out of view by use of swipe gestures.

Beside allowing us to validate their existence and usefulness, as well as giving a preview of what kind of interaction patterns to expect, this first experiment also gave insight into some challenges which need to be dealt with when implementing the full mining process. First, we found that without additional filtering, the results are largely dominated by itemsets comprised of packages belonging to a number of Android standard libraries. While this problem could easily be remedied by filtering out all sub-packages of `java.*, android.*, com.google.*`, this would also rule out any interaction patterns between third-party libraries and standard libraries, such as the one shown in Listing 3. Therefore, alternative ways of filtering the resulting patterns need to be evaluated. Second, while pre-compiled Android applications, *apks*, facilitate automated analysis due to their well-formed and fixed structure, it turned out that many of the analyzed apps obfuscate their generated byte-code, which can lead to false-positives if this is not dealt with in the trace generation step.

4 RELATED WORK

While in recent related work only little attention was paid to the study of API interaction patterns, in this section we give an overview of other approaches which study the interplay between multiple APIs or the analysis of interactions between other domain concepts, e.g., feature interaction.

API Specification Mining and API Recommendation. Inferring valid API specifications is a critical challenge since false-positive patterns provide little value, may hinder developers, or even cause errors, e.g., when suggesting inappropriate code completions. Thus, the researches in this domain strive to find methods for improving the quality of inferred patterns [6, 10].

However, usually, these approaches concentrate on single APIs. Nevertheless, in the field of Web APIs some of the latest research considers recommendations systems for simplifying the composition of web APIs - so-called mashups. In particular, developers should be equipped with a means of finding Web APIs that both fit their search criteria and are applicable in the concrete context. Here, different approaches find co-invocations of APIs by means of analyzing historical API usage and co-occurrence data [9, 13].

Thung et al. suggested a general approach for recommending APIs and libraries. For this purpose, they combined association rule mining on libraries with a nearest-neighbor approach [12]. In contrast to interaction patterns, this approach only recommends *which* libraries to use, but not *how*.

Lastly, Kula et al. analyzed the dependencies of APIs and noticed that these are rarely updated [5]. They state that it becomes much harder for software developers to migrate and update their respective APIs and libraries. Thus, understanding API interactions, and how they change with time, is an interesting question for the analysis of API interaction patterns.

Feature Interaction. The study of interactions is not restricted to APIs or to the specification mining domain. In software product lines (SPL) research, independent software behaviors - so-called features - are combined in order to a form a new product. One

```
public class CustomGlideModule implements GlideModule {
    @Override public void registerComponents(Context context, Glide glide) {
      OkHttpClient client = new OkHttpClient();
      OkHttpUrlLoader.Factory factory = new OkHttpUrlLoader.Factory(client);
      glide.register(GlideUrl.class, InputStream.class, factory);
    }
}
```

Listing 2: Example Interaction between `okhttp3.OkHttpClient` and the `glide` image loading library

```
SlidingMenu sm = new SlidingMenu(Context context);
sm.attachToActivity(Activity activity, SlidingMenu.SLIDING_WINDOW | SlidingMenu.SLIDING_CONTENT);
```

Listing 3: Interaction Pattern between `android.app.Activity` and `com.jeremyfeinstein.slidingmenu.lib.SlidingMenu`

major challenge in this domain is understanding and dealing with feature interactions. These represent new behaviors which only become apparent when a specific combination of individual features is present [3]. Since feature interactions are not always predictable they may result in positive, e.g., performance improvements [8], or negative effects, e.g., security issues, [11]. Thus, SPL research is developing techniques to automatically discover feature interactions [2, 11]. Synergies between API interaction patterns and feature interactions may be another interesting research direction for further work.

5 CONCLUSION

In this idea paper we propose a method for the systematic extraction and analysis of API interaction patterns. These can be useful for a number of automated software engineering tasks, including reverse-engineering, documentation, and various recommender systems.

Our process begins with a de facto standard specification mining step, based on static, inter-procedural trace generation and sequence mining, followed by two filtering steps. First, we separate interaction patterns from usage patterns referring to only a single API. Then, we further discriminate between two types of interaction patterns, *local* and *workflow*, based on a cohesion metric. Our hypothesis is that local patterns show how two or more APIs interact directly within source code, making them useful for use-cases such as source code completion. In contrast, we expect workflow patterns to describe interactions on a more global level, e.g. how typical library combinations for data acquisition, analysis and visualization are configured to work well for a specific application.

We performed a preliminary investigation of interaction patterns on a set of nearly 500 Android applications of different sizes and from various domains, which will also serve as case study for the upcoming analysis. The initial results prove at least the existence of meaningful *local* interaction patterns. In particular, we found valuable patterns relating to different use-cases including *io, media,* and *gui*. The main drawback of the method used in the preliminary investigation is that, while it is easy to manually spot *local* interaction patterns, finding substantial evidence of the anticipated *workflow patterns* proved more complex. Therefore, we are very interested in whether or not the combination of inter-procedural trace generation and pattern-cohesion filtering will yield the expected results.

REFERENCES

[1] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining Specifications. In *29th POPL*. ACM, 4–16.

[2] Sven Apel, Alexander von Rhein, Thomas Thüm, and Christian Kästner. 2013. Feature-Interaction Detection Based on Feature-Based Specifications. *Computer Networks* 57, 12 (2013), 2399 – 2409. https://doi.org/10.1016/j.comnet.2013.02.025 Feature Interaction in Communications and Software Systems.

[3] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. 2008. What's in a Feature: A Requirements Engineering Perspective. In *11th FASE*, José Luiz Fiadeiro and Paola Inverardi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 16–30.

[4] Philippe Fournier-Viger, Antonio Gomariz, Ted Gueniche, Azadeh Soltani, Cheng-Wei Wu, and Vincent S Tseng. 2014. SPMF: A Java Open-Source Pattern Mining Library. *JMLR* 15, 1 (2014), 3389–3393.

[5] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do Developers Update Their Library Dependencies? *EMSE* 23, 1 (01 Feb 2018), 384–417. https://doi.org/10.1007/s10664-017-9521-5

[6] Claire Le Goues and Westley Weimer. 2009. Specification Mining with Few False Positives. *15th TACAS* (2009), 292–306.

[7] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017. Bayesian Specification Learning for Finding API Usage Errors. In *11th Joint ESEC/FSE*. ACM, 151–162.

[8] Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding Near-optimal Configurations in Product Lines by Random Sampling. In *11th FSE (ESEC/FSE 2017)*. ACM, New York, NY, USA, 61–71. https://doi.org/10.1145/3106237.3106273

[9] Svetlana Omelkova and Peep Küngas. 2016. Personal Web API Recommendation Using Network-based Inference. In *4th SALAD@ESWC*.

[10] Martin P Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. 2013. Automated API Property Inference Techniques. *TSE* 39, 5 (2013), 613–637.

[11] Larissa Rocha Soares, Pierre-Yves Schobbens, Ivan do Carmo Machado, and Eduardo Santana de Almeida. 2018. Feature Interaction in Software Product Line Engineering: A Systematic Mapping Study. *Information and Software Technology* 98 (2018), 44 – 58. https://doi.org/10.1016/j.infsof.2018.01.016

[12] Ferdian Thung, David Lo, and Julia Lawall. 2013. Automated Library Recommendation. In *20th WCRE*. IEEE, 182–191.

[13] Lina Yao, Xianzhi Wang, Quan Z. Sheng, Boualem Benatallah, and Chaoran Huang. 2018. Mashup Recommendation by Regularizing Matrix Factorization with API Co-Invocations. (2018), 1–1. https://doi.org/10.1109/TSC.2018.2803171