A Vision for Helping Developers Use APIs by Leveraging Temporal Patterns

Erick Raelijohn
Université de Montréal
erick.raelijohn@umontreal.ca

Michalis Famelis
Université de Montréal
famelis@iro.umontreal.ca

Houari Sahraoui
Université de Montréal
sahraouh@iro.umontreal.ca

Abstract—To achieve any meaningful task of a certain complexity, developers need to use APIs. Learning to use them is both time consuming and cognitively demanding. We propose to leverage a formal description of API usage as temporal patterns to help developers make sense of the complexities of working with APIs. To achieve this, we propose to deploy recommender systems at various points of the development process that make these patterns useful when most needed. In this paper, we illustrate the approach on a non trivial, real world running example from Android development. The example allows us to articulate a research agenda for leveraging API usage patterns during: (a) testing and compilation times by recommending potential violations of the patterns; (b) coding time by recommending API method calls; and (c) API delivery by recommending improvements to documentation.

I. Introduction

To realize complex tasks developers need to reuse multiple software libraries in form of APIs. To master them, developers need to expend considerably energy to learn and understand a set of often complex usage constraints that bound their usage. In general, learning an API requires reading and understanding its documentation, looking at examples, and searching for community support online. That may consequently reduce developers productivity and even affect the quality of the code they are writing, especially when the APIs are poorly documented or not well understood by online communities.

We want to help developers best use APIs regardless of the quality of its documentation. To accomplish that, we want to use a formal description of APIs in form of Linear Temporal Logic (LTL). In other words, the latter represents a usage pattern of the API. It is important to say that at this phase, we do not make any assumptions regarding the provenance of usage patterns. They could for example be written manually [1], mined from traces [2], or synthesized [3].

The goal is to leverage those patterns to deliver recommendations during multiple phases of development:

a) Testing and compilation phases: we could verify the code validity by analyzing its execution trace. In the case of compilation, we can generate pseudo-traces by building a control flow graph (CFG) to compute all possible execution path. Then, we could recommend a potential violation of a pattern which is a potential misuse of all concerned API methods.

b) Coding phase: In this phase, we want to deliver feedback to developers by recommending API method call and also generating warning on the fly in case of a potential violation of a pattern. The process differs from the testing and compilation phases because this time developers are still writing the code which means we will need to compute execution paths on an Incomplete CGF (ICGF).

c) API delivery by providing documentation supplements: LTL patterns are a formal description that can be interpreted as rules. In our case, the rules refer to a relation between multiple API methods. If a pattern is not too complex it can be translated into natural language, which can be used to provide documentation with clear rules about how a set of methods should be used together.

Using recommendations for API usage is not a new idea. Different approaches were proposed for different degrees of usages. At an early stage, recommending systems may help to select, for a particular need, a suitable API [4] or a specific API method [5]. Such systems may also help developers by mining or synthesizing usage examples [6] or by recommending parameters for method calls [7]. The novelty of our vision is in the use of temporal patterns for the correct orchestration of API methods.

We illustrate our vision using a motivating example in Section II, outline our research agenda in Section III, and conclude in Section IV.

II. Motivating Example

To illustrate our proposed approach, we use the example of a development team that is building the Android application Actrack that collects all physical related data (steps, heart rate, sleep tracking ect...) of the user to draw different statics so that the user can keep track of their progression by looking in the app. The app Actrack requires interacting with a persistent storage, and therefore the team must use the Android SQLite API[1]. In the course of development, the team has produced the code for the Actrack class `CrHelper`, shown in Figure 1. In the rest of the paper, we call code like this "client code", to differentiate it from API code.

The client code of the class `CrHelper` depends on `SQLiteClosable`. We see in line 56 that they invoking

[1]https://developer.android.com/reference/android/database/sqlite/package-summary

```
10  public class CrHelper extends SQLiteClosable {
11      private WeakHashMap<SQLiteClosable, Object> mPrograms;
12      Map<String, SQLiteCompiledSql> CQueries = Maps.newHashMap();
13  ›   public void onCreate(SQLiteDatabase db) {…
20  ›   public int update(Uri uri, ContentValues values, String where, String[] whereArgs) {…
49
50      public void close() {
51            Iterator<Entry<SQLiteClosable, Object>> iter = mPrograms.entrySet().iterator();
52              while (iter.hasNext()) {
53                Map.Entry<SQLiteClosable, Object> entry = iter.next();
54                SQLiteClosable program = entry.getKey();
55                if (program != null) {
56                    program.onAllReferencesReleasedFromContainer();
57                }
58              }
59      }
60  }
61
```

Fig. 1. `close()` method

the method `onAllReferencesReleasedFromContainer()` which releases all references to the object and under certain circumstance close the database if the last connection is removed. In our scenario we assume that the developers are not very experienced with the Android SQLite API and that the documentation is not enough for them, as it is not stated explicitly that an error will occur if a database is closed even though SQL statements are not finalized.

In a situation like this, the development team would spend a considerable amount of time gaining experience on using the API. This is time that could instead be put towards improving Actrack. In other words, it would be desirable to help the team use the API correctly, even though they might not have mastered all aspects of it. We envision creating tool support that would allow the development team to take advantage of previously amassed experience about the correct usage of the API, without needing to rely on incomplete documentation or on extensive web searches and word of mouth information scraped from the internet.

In our example, we know from experience that SQLite throws an exception if there are non finalized statements, such as in `CQueries` line 12 in Figure 1. To avoid that, all compiled SQL statements should be de-allocated using the database `SQLiteCompiledSql.releaseSqlStatement()` method. We envision encoding this experience in a way that can be leveraged automatically by the development team. One way that allows taking advantage of sophisticated automated reasoning techniques is to describe such experience about the correct usage of APIs as patterns. If we encode our experience about correct usage of the Android SQLite API in Linear Temporal Logic (LTL) using the precedence pattern from Dwyer work (Paper [1]) we get the following pattern P:

$$\neg(o \vee c \vee rr) W rs$$

The aliases o, c, rr, rs in P are explained in Table I. W is the LTL operator "weak until", thus P effectively says that one of the three methods o, c, rr has to be called at least until the method rs is called; if the method rs is never called, then the one of these three methods must continue to be called. (We assume that the pattern only applies to API calls, ignoring all other code.)

TABLE I
Variable description

Variable	API Method
o	`onAllReferencesReleasedFromContainer`
c	`close`
rr	`onAllReferencesReleased`
rs	`releaseSqlStatement`

This encoding allows us to envision various approaches to deploy automated techniques for assisting the development team correctly use the API. We outline them below:

a) Documentation completion: We propose to help the team during the development phase, by improving the available documentation for the Android SQLite API. For example, we can generate a description of the pattern P in natural language, as follows:

> All compiled SQL statements should be de-allocated from the cache by using the `releaseSqlStatement()` method before closing the concerned database.

Then this can be integrated with existing documentation of the SQLiteClosable class[2].

b) Compilation and Testing phase: We can also use P to help the team during compilation and testing time. In our example, the code in Figure 1 generates no errors or warnings at build time, because it is syntactically correct. Instead, we envision providing a warning at compile time to the developers that the API usage pattern P is not satisfied. An example of such a warning is shown in Figure 3 where the user is alerted that using the method `onAllReferences()` could close the database, generating an error if there are any non-finalized statement. The developers would then get the opportunity to investigate how to change their code in order to fix any potential issues.

Furthermore, given that the developers do not have extensive knowledge about how to correctly use the Android SQLite API, it is unlikely that they would write a test suite that would cover all problematic cases, such as the ones cause by not deallocating compiled SQL statements. We envision using pattern P during testing time as well to automate the creation of complete test suites.

Using functionalities such as these, the development team would be able to produce a version of the code that correctly uses the API, i.e., does not violate the pattern P. Such a version is shown in Figure 2, where the `releaseSqlStatement()` method is appropriately invoked in line 55.

c) Code completion: Consider the scenario where Dana, an individual developer of the Actrack team is in

[2]https://developer.android.com/reference/android/database/sqlite/SQLiteClosable

```
10  public class CrHelper extends SQLiteClosable {
11      private WeakHashMap<SQLiteClosable, Object> mPrograms;
12      Map<String, SQLiteCompiledSql> CQueries = Maps.newHashMap();
13  ›   public void onCreate(SQLiteDatabase db) {⋯
20  ›   public int update(Uri uri, ContentValues values, String where, String[] whereArgs) {⋯
49
50      public void close() {
51            Iterator<Entry<SQLiteClosable, Object>> iter = mPrograms.entrySet().iterator();
52            try {
53              synchronized (CQueries) {
54                  for (SQLiteCompiledSql compiledSql : CQueries.values()) {
55                      compiledSql.releaseSqlStatement();
56                  }
57                  CQueries.clear();
58              }
59              while (iter.hasNext()) {
60                Map.Entry<SQLiteClosable, Object> entry = iter.next();
61                SQLiteClosable program = entry.getKey();
62                if (program != null) {
63                    program.onAllReferencesReleasedFromContainer();
64                }
65              }
66
67            }  finally {
68                  onAllReferencesReleased();
69                  unlock();
70             }
71    }
```

Fig. 2. A correct variation of `close()` method

```
10  public class CrHelper extends SQLiteClosable {
11      private WeakHashMap<SQLiteClosable, Object> mPrograms;
12      Map<String, SQLiteCompiledSql> CQueries = Maps.newHashMap();
13  ›   public void onCreate(SQLiteDatabase db) {⋯
20  ›   public int update(Uri uri, ContentValues values, String where, String[] whereArgs) {⋯
23
24      public void close() {
25
26            Iterator<Entry<SQLiteClosable, Object>> iter = mPrograms.entrySet().iterator();
27            try {
28              synchronized (CQueries) {
29                  for (SQLiteCompiledSql compiledSql : CQueries.va
30                      compiledSql.releaseSqlStatement();
31                  }
32                  CQueries.clear();
33              }
34            while (iter.hasNext()) {
35              Map.Entry<SQLiteClosable, Object> entry = iter.next(
36              SQLiteClosable program = entry.getKey();
37              if (program != null) {
38                  program.onAllReferencesReleasedFromContainer();
39
40
```

#onAllReferences... could close the database and generate an error if there are any unfinalized statement
quickfix:
add releaseSqlStatement()

Fig. 3. Recommendation example `close()`

the process of developing the class `CrHelper`. Assume that Dana is working in her favourite IDE

We envision to help Dana by integrating into her IDE a recommender system that uses patterns like P to propose to her code completion recommendations that guide her towards correctly using the API. As described in the previous paragraph, we envision generating build-time warnings for incorrect API usage. It is conventional in modern IDEs for the build system to be invoked incrementally in the background any time the developer saves the contents of a source code file. In our example, when Dana finished writing the method `onAllReferencesReleasedFromContainer()`, the build system was invoked. Assuming she has in her disposal tooling such as that described before, Dana would get a warning at line 38, as shown in Figure 3. Specifically, the code is underlined to show that there is a potential violation of the pattern P. The "quick fix" mechanism available in many modern IDEs, allows Dana to hover over the line, and access a panel that explains the warning and gets a recommendation for repairing the problem. In our example, we illustrate the case where the "quick fix" of Dana's IDE proposes a way to naively satisfy P by invoking the method `releaseSqlStatement()`.

III. Research Agenda

We now discuss our research agenda for leveraging APIs usage pattern to help developers. The overarching research question is: (RQ1) how can LTL patterns describing API usage be best exploited to provide useful information for the developer? Our approach is to use them to produce recommendations for the developers during multiple phases as explained in the next subsections.

Several researchers have worked in the past on extracting APIs usage pattern. Generally, the different methods differ from each other in terms of the template they use to build the pattern. Yang et al. [8] and Uddin et al. [9] used ordered sets of co-used methods while other researchers [10]–[12] used a set of unordered co-used methods. Pattern can also be synthesised [3], written manually using predetermined pattern [1] or mined from trace log [2].

To the best of our knowledge, there exists no work on leveraging temporal patterns for generating recommendations to assist developers better use APIs. Nonetheless, Our approach is independent and complementary to previous work and can be applied to API usage patterns irrespective of their provenance. Further, we envision closing the loop by integrating recommendation usage statistics into pattern mining approaches in order to better guide the discovery process.

A. Testing and Compilation Phase

Developers should be able to leverage the patterns during compilation and testing time, thus complementing compiler messages and existing test suites with additional knowledge about using APIs of interest. This would allow them to find problems that would otherwise be hard to detect without targeted static analysis. Our approach is to help them determine the method or at least the part of the code where they should focus their efforts to correctly use an API of interest.

We propose to do this by performing a "check-up" of client code for conformity with correct usage patterns of APIs at compile and testing time. We thus pose the question: (RQ2) How can we check a pattern for a given client code? We envision doing this formally for a set of LTL patterns, using execution traces of the client code. We can then verify that the client code's invocation of API methods respect the given patterns. However, the obvious problem is that there are no execution traces available at compilation time. Thus: (RQ3) How can we check a pattern without executing the client code? A potential approach is to generate "pseudo-traces" from the control flow graph of the client code, that approximate possible execution paths.

(RQ4) How can developers safely interpret our recommendation? The challenge is to provide clear, useful and concise recommendation when we detect potential API misuse. In our example, the correction to the misuse (shown in Figure 2) is relatively complicated and requires multiple inputs. A further helpful step could be to integrate existing approaches [4], [6] to demonstrate examples of API usage, as well as pointers to online resources for further consultation. This can also help in cases where the pattern is complex and hard to understand.

Since our approach is based on the usage of general patterns of API usage it is conceivable that we would be generating warnings for uncommon, albeit benign usages of API methods. Further, if our recommendations are based on approximations of possible execution paths with the "pseudo-trace" approach described above, we could also generate false positives because of the loss of information involved in the approximation. Generally, we pose the question: (RQ5) How can we reduce false positive recommendations?

Another challenge is to handle the case where API usage is spread across multiple methods of client code: (RQ6) How can we analyze API usage across different methods in client code? This would require some more sophisticated static analysis techniques [13] that would allow us to reconstruct inter-procedural approximations of execution.

B. Coding Time

We want to use the patterns to help the developers as they are coding. Thus, (RQ7) How can we check a pattern on incomplete code? If we keep the same approach of computing possible execution traces, we would need to do that whenever something is added to the code which will require a lot of resources. Thus, (RQ8) How can we compromise the need to provide on the fly recommendations with the generally more heavyweight formal techniques required to process patterns? We envision adapting runtime monitoring and incremental repair techniques [14], combined with appropriate domain assumptions.

Following that, (RQ9) How could we provide useful recommendation during coding phases? In our example, recommending a completion to Dana is non-trivial, and hard to fully explain in a pop-up. Providing recommendations becomes more difficult when multiple patterns are involved. Providing an explanation and recommendation to every single pattern would be overwhelming and too invasive. We thus need to order recommendations according to usefulness to the developer. Thus, (RQ10) How can we determine the most relevant pattern each time we want to generate a recommendation? A potential approach could be to crowdsource this, e.g., using a public repository of API usage patterns.

C. Documentation improvements

Writing and understanding LTL is challenging, as it requires a high level of expertise in formal logic. It is thus useful to translate LTL patterns into natural language that is easier for humans to comprehend. (RQ11) How can we describe LTL API usage patterns into natural language useful to developers? Existing approaches typically take an approach based on using predetermined sets of patterns [1], [15]. However, since API usage patterns can also be mined automatically, we will need pattern independent techniques. One approach is to use techniques for machine learning and example-based translation [16]. A further challenge is to integrate generated descriptions of patterns with existing API documentation.

IV. Conclusion

Learning how to correctly use an API is difficult. We have presented a vision for supporting developers in this task that relies on the formal specification of API usage scenarios as LTL patterns. We have then outlined a research agenda for deploying formal techniques to generate recommendations at different stages of software development.

Although our approach targets multi-methods temporal patterns for API usage, it can be complemented with method level usage constrains like ones described in [17]. This allows to cover a large spectrum of misuse situations.

References

[1] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in Proc. of ICSE '99.

[2] C. Lemieux, D. Park, and I. Beschastnikh, "General ltl specification mining (t)," in Proc. of ASE'15.

[3] M. A. Saied, H. Sahraoui, E. Batot, M. Famelis, and P.-O. Talbot, "Towards the automated recovery of complex temporal api-usage patterns," in Proc. of GECCO'18.

[4] M. A. Saied, A. Ouni, H. Sahraoui, R. G. Kula, K. Inoue, and D. Lo, "Improving reusability of software libraries through usage pattern mining," Journal of Systems and Software.

[5] C. Xu, X. Sun, B. Li, X. Lu, and H. Guo, "Mulapi: Improving api method recommendation with api usage location," Journal of Systems and Software.

[6] R. P. Buse and W. Weimer, "Synthesizing api usage examples," in Proc. of ICSE'12.

[7] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou, "Automatic parameter recommendation for practical api usage," in Proc. of ICSE'12.

[8] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining Temporal API Rules from Imperfect Traces," in Proc. of ICSE '06.

[9] G. Uddin, B. Dagenais, and M. P. Robillard, "Temporal analysis of api usage concepts," in Proc. of ICSE'12.

[10] M. Gabel and Z. Su, "Javert: Fully automatic mining of general temporal properties from dynamic traces," in Proc. of FSE'08.

[11] M. A. Saied, H. Abdeen, O. Benomar, and H. Sahraoui, "Could we infer unordered api usage patterns only using the library source code?" in Proc. of ICPC'15.

[12] M. A. Saied, O. Benomar, H. Abdeen, and H. Sahraoui, "Mining multi-level api usage patterns," in Proc. of SANER'15.

[13] S. H. Jensen, A. Møller, and P. Thiemann, "Interprocedural analysis with lazy propagation," in Proc. of SAS'10.

[14] J. Simmonds, Y. Gan, M. Chechik, S. Nejati, B. O'Farrell, E. Litani, and J. Waterhouse, "Runtime monitoring of web service conversations," IEEE Transactions on Services Computing, vol. 2, no. 3, pp. 223–244, 2009.

[15] R. Nelken and N. Francez, "Bilattices and the semantics of natural language questions," Linguistics and Philosophy, vol. 25, no. 1, pp. 37–64, 2002.

[16] H. Somers, "Review article: Example-based machine translation," Machine Translation, vol. 14, no. 2, pp. 113–157, 1999.
[17] M. A. Saied, H. Sahraoui, and B. Dufour, "An observational study on api usage constraints and their documentation," in International Conference on Software Analysis, Evolution and Reengineering, 2015, pp. 33–42.