Feature Location Benchmark for Decomposing and Reusing Android Apps

Yutian Tang[1], Hao Zhou[1], Zhou Xu[2], Xiapu Luo[1], Yan Cai[3], and Tao Zhang[4]

[1]Department of Computing, The Hong Kong Polytechnic University
[2]School of Big Data and Software Engineering, Chongqing University
[3]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[4]Faculty of Information Technology, Macau University of Science and Technology
[*]Corresponding Author: csxluo@comp.polyu.edu.hk

Abstract—**Software reuse enables developers to reuse architecture, programs and other software artifacts. Realizing a systematical reuse in software brings a large amount of benefits for stakeholders, including lower maintenance efforts, lower development costs, and time to market. Unfortunately, currently implementing a framework for large-scale software reuse in Android apps is still a huge problem, regarding the complexity of the task and lacking of practical technical support from either tools or domain experts. Therefore, proposing a feature location benchmark for apps will help developers either optimize their feature location techniques or reuse the assets created in the benchmark for reusing. In this paper, we release a feature location benchmark, which can be used for those developers, who intend to compose software product lines (SPL) and release reuse in apps. The benchmark not only contributes to the research community for reuse research, but also helps participants in industry for optimizing their architecture and enhancing modularity. In addition, we also develop an Android Studio plugin named caIDE for developers to view and operate on the benchmark.**

Index Terms—**Software product line, feature location benchmark, reuse, android apps**

I. BACKGROUND

Mobile apps dominate our daily life nowadays. Users can easily download the apps from online app stores, such as, Google Play, Amazon, Apple App Store and Windows Phone marketplace. According to a recent report from AppBrain [1], there are over 2,850,172 apps publicly available on Google Play until the fall of 2018. The amount of apps increased more than fourfold between 2013 and 2018.

The huge market with a large number of active users attacks developers to contribute and release their artifacts via app stores. According to the study conducted on 4,323 apps, software reuse is the common practice among app developers. For all apps concerned, 61% of the `classes` appears in two on more apps on average. Furthermore, currently reuse practice in apps mainly focuses on API library reuse, class reuse and inheritance [1], [2]. Unfortunately, the current reuse practice and tools cannot help developers in implementing a systematically reuse of software artifacts in terms of app development. Therefore, the current practice cannot help developer reuse software artifacts systematically. Such as, implementing

a software product line (SPL) in Android apps. A software product line is a set of software-intensive systems that share a common set of features. Each system in the product line contains unique segments that are defined to fulfill specific needs of a particular market [3]. Then, developers will create the product variants from the product line and tailor the variant to end users.

By reusing the component systematically, stakeholders can benefit from the software product line, including reducing development efforts, reducing maintenance efforts, and time to market [3]. Unfortunately, based on some existing research experience [1], [2], [4], [5], existing reuse practice in apps does not follow a framework with a theoretical foundation. Instead, developers are more likely to follow a naive "copy-paste" strategy, which may misuse some functions and even introduce potential security problems.

To contribute the reuse practices in software product line and further help stakeholder reuse artifacts in apps, in this work, we take the first step to the systematical reuse. That is, we present an Android Studio plugin for developers to conduct the feature location, and release a benchmark of feature location. A feature in an SPL represents an user-visible function, aspect or characteristic of a system [3]. Feature location is an important and fundamental procedure in software reuse. The task of feature location is trying to find mapping between code fragments and features [6]. With the feature location benchmark, more research prototypes can be proposed and tested, especially the reuse framework for apps. This can help developers get arid of immature reusing practices. Also, due to the different context from desktop applications, developers, who intend to focus on proposing reuse framework for apps, should mainly reference the feature location benchmark that targets on apps.

Contribution. To help developers boost their reuse practice and provide back-end supports for reusing frameworks, we provide a benchmark that contains over five hundred open sourced android apps on F-Droid[2]. Specifically, in this work, we mainly make the following contributions.

- We release a feature location benchmark for Android apps. The benchmark contains over five hundred apps

---

from F-Droid.

- We propose an Android Studio Plugin named caIDE for the benchmark. Developers can explore and even edit the benchmark with the tool provided. Moreover, developers can also use our tool to build their own benchmark and release it to the community.
- We also discuss the uniqueness issues in feature location for apps and differences between apps and desktop applications in terms of software reuse.

> The benchmark of 198 Android apps with ground-truth, the scenarios, and caIDE tool can be found at:
> **https://sites.google.com/view/caide**,
> including technical tutorial on how to use it.

Skeleton This paper is organized as follows: Section II presents the overview of benchmark construction and the caIDE tool. Section III and IV introduce the technical details on how to extract features, build feature models and locate features, respectively. Furthermore, Section V introduce the details of the benchmark in terms of features, feature models, and feature annotations. In addition, our tool provides several additional functions to allow developers further operate on the benchmark. This part is covered in Section VI. We conclude this work in Section VIII.

II. BENCHMARK CONSTRUCTION AND *caIDE*

To realize a systematic reuse in a system, the software reuse community normally propose *bottom-up* techniques. Typically, in a *bottom-up* approach, there are three major objectives: feature identification (Section III-A and III-B), feature location (Section IV-A and IV-B), and re-engineering [7], [8]. Specifically, feature identification aims at finding and discovering features for the product line. Then, in the feature location, the feature and its implementations are mapped. For example, in a music app (e.g. JOOX music), *play a music* could be a feature. The feature location approaches will find the code segments in the code base that corresponds to this feature. Finally, feature re-engineering is a transformation process that transforms the annotated system into target products. As for our benchmark, we only cover the *feature identification* and *feature location* process. As for the feature re-engineering, it is totally a customized procedure. That is, stakeholders and domain experts have to design the configuration for target products. However, caIDE also provide additional functions to help stakeholders on this procedure automatically.

In general, caIDE provides following major usages: (1) annotating features in the code base; (2) viewing the annotations; and (3) other supporting functions (Section VI). All three objects in a typical *bottom-up* process are covered in caIDE. Specifically, *feature identification* and *feature location* are covered in the annotation process. The *re-engineering* process is described in supporting functions in caIDE (Section VI).

Annotation Process. The caIDE provides a series of functions for developers and domain experts to define features, feature model and annotate the features with visual supports. To annotate a product, caIDE guides developers through the following steps.

- **STEP 1.** Developers have to define the features and the feature model in the product line. (Section III)
- **STEP 2.** Developers have to assign each feature a unique background color. Here, caIDE can automatically assign each feature a color. Developers can still customize the background color.
- **STEP 3.** Developers can set code fragments with features. (Section IV)

For step 1, caIDE provides a visual support for developers to draw the feature model and edit the features in the feature model easily. After this step, a feature model file named `featuremodel.afm` is built. The feature model file strictly follows the feature model grammar defined by Don Batory [9], which is a well-adopted feature model representation. Then, caIDE will guide developers to assign unique color to each feature. Later, developers can assign the code fragments to features as defined in step 3. Specifically, the developers first select a code range in the editor in Android Studio IDE, then they can set the feature for the code range from the context menu. caIDE will visit the AST of the file and get the AST nodes within the code range. Then, all these nodes will be assigned to the features. Note that, if there is a "parent-child" relation between two AST nodes in the range, we only annotate the parent AST node. Once we remark a parent AST node to a feature, all its children are annotated the feature as well. The annotation will also be stored to external files (named <file>.color) for reusing and displaying the background color. For example, when we annotate a code segment in file `Reader.java`, then the caIDE will create a new file named `Reader.color` to store the annotation information for the file `Reader.java`. By putting those annotation files into the same directory with its source code, caIDE avoids the naming issue. We will introduce the corresponding technical details in Section III and IV.

Explore Process. caIDE helps stakeholders check the annotated systems. To display the benchmark, caIDE helps developers collect all information from the annotation files(`.color`) in the program and renders the code fragments with background colors. More specific, caIDE will first search the `featuremodel.afm` to recover the feature model. Then, it will check the `color.json` file to build the mapping between features and colors associated. At last, caIDE will inspect all source code files and all annotation files to establish the mapping between AST nodes and features. When a user opens a file in the editor, the corresponding AST nodes will be remarked with background colors.

Far more than these fundamental solutions, we extend caIDE by providing more possible actions for developers to directly use the benchmark we provided with ease. The details of these functions are introduced in Section VI.

III. FEATURE AND FEATURE MODEL

A. Feature and Feature Identification

Features in an app describe the main functions or services provided by the app, which are normally visible for end-users. As defined previously, feature names of an app should be defined by the domain experts. Normally, a domain expert has

to define the features and design the feature model. However, this procedure could be time-consuming and tedious. We provide an automatic approach to recommend some features for developers. We extract the features from: (a) app's description on Google Play Store; (b) app's description for its open source repository; and (c) textual information from app's implementation. Specifically, for (c), we extract the identifiers from the code based to explore the potential feature names in the app.
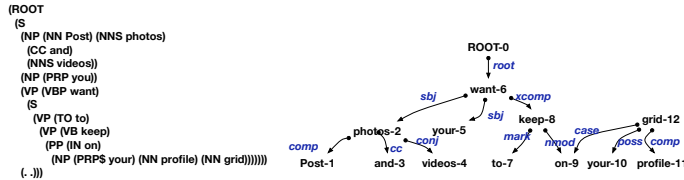

Fig. 1: Parse tree of the sentence and universal dependencies between words in the sentence

We use parts of speech (POS) tagging and parsing, to explore the architectures of sentences. POS tags are assigned to a single word according to its role in the sentence. Commonly-used POS tags include ADJ (i.e., adjective), VB (i.e., verb), NN (i.e., noun). Given the input sentence, we use the Stanford Parser to get the parse tree of the input sentence and the universal dependencies between words. The parse tree contains the verb phrases (i.e., VP) and noun phrases (i.e., NP) of the sentences and POS tags of each word. The universal dependency describes the relationship between words. For example, sbj refers to subject. We explore the parse tree and universal dependencies between words to identify the noun phrases and verb phrases contained in the sentence. These phrases are considered as possible features.

Specifically, for (c), we treat each `class` as flat text and use tf-idf to compute the importance of terms. The terms are ordered to be recommended as feature names. At last, feature names are selected from these recommendations.

B. Building Feature Model

With the features and feature annotations collected from the previous step, we build the feature model. Technically, the feature model should be built by domain experts of the systems. In caIDE, the feature model can be built manually. However, to reduce the bias and provide a handful approach for the case that the domain expert is not available, basically we follow She et al.'s work to build the feature model [10] in the benchmark. She et al.'s approach [10] requires two inputs: the complete dependencies and extensive descriptions. The way to generate these two inputs is described as follows.

- **Dependencies**: the dependencies are extracted from two aspects: (1) we manually learn the app's description and user manual (if any) and extract the potential dependencies; (2) we install the app on the emulator and try each feature in the app. Then, we describe the relations between features.
- **Descriptions**: the descriptions of features are collected from two parts: (1) the description, wiki and user manual

of the app; (2) some descriptions of features are presented in the project's change log and commit message. When developers use some version control tools (e.g. svn, github) to manage their project, developers have to write a commit message to describe the changes for each commit.

With these information provided, the feature model can be built automatically with the approach proposed in [10].

Example. In the app AnkiDroid, our approach recommends 12 features, including *T2T* (text-to-speach), *CardBrowser*, *Statistics*, *NightMode*, *FullBackup*, *Syncing*, *WriteAnswers*, *WhiteBoard*, *DictionaryIntegration*, *CardEdit*, *Import*, and *CustomFont*. The feature model is built with the feature model construction approach, as shown in Figure 2.
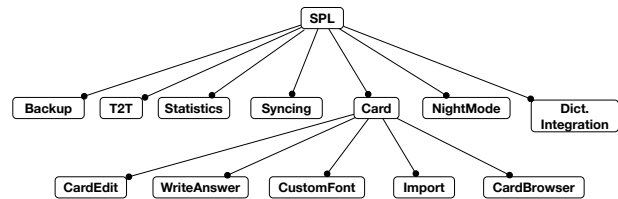

Fig. 2: The feature model of AnkiDroid

IV. FEATURE LOCATION

A. Annotation Scheme

Prior to feature location techniques, we first introduce the annotation scheme we used in the caIDE. Our benchmark is created with our plugin named caIDE, which is built with IntelliJ and targeted at Android Studio and IntelliJ. With the caIDE, we can decompose the app into features, which may have a fine granularity. Developers first start with a fully composed app with all features implemented in the application. In software product line engineering, such a system is called legacy application. Then, developers can annotate code fragments with different features. One code can be associated with one or more features. To annotate code segments with features, currently there are two commonly used approaches: colored annotation [11] and precondition compiling based annotation (a.k.a #ifdef directive) [12]. Specifically, the colored annotation binds code fragments with different background colors. Each color represents a unique feature defined in the feature model.

Example. As shown in Listing 1, different colors are associated with different features. Code fragments in the program are rendered with different colors to represent the feature annotations. Apparently, there are three features involved in the running example, include `push`, `pop`, and `lock`. In caIDE, we also adopt this annotation scheme.

```
1  class Stack{
2    int size = 0;
3    Object[] elementData = new Object[maxSize];
4    boolean transactionsEnabled = true;
5    void push(Object o){
6      Lock l = lock();
7      elementData[size++] = o;
8      unlock(l);
9    }
```

```
10   Object pop(){
11      Lock l = lock();
12      Object r = elementData[--size];
13      unlock(l);
14      return r;
15   }
16   Lock lock(){
17      if (!transactionsEnabled) return null;
18      return Lock.acquire();
19   }
20   void unlock(Lock lock){/*...*/}
21   String getLockVersion() { return "1.0";}
22 }
23 class Lock {/*...*/}
```

Listing 1: Sample of Colored Annotation

Whereas, precondition based annotation is mainly used in product line, that developed in C/C++. In practice, developers use feature names as pre-conditions. If and only if the feature is selected, the code fragment in the directive can be covered in the target system. For example, when we use Linux system, we can configure the kernel by selecting the features we want and disabling features we are not interested.

Example. In List. 2, there are two preconditions: `CONFIG_SMP` and `CONFIG_APIC`. The code `block 1` will be executed if the macro `CONFIG_SMP` is defined. The execution condition for code `block 2` is `!CONFIG_SMP && CONFIG_APIC`. Therefore, some benchmarks use this annotation strategy to remark the features by setting the features as the preconditions. Then, when some features are disabled in the configuration, the `if` condition (`#ifdef CONF`) will be `FALSE`. The code block within the condition will not be covered in the product built.

```
1 #ifdef CONFIG_SMP
2 //block 1
3 #elif defined CONFIG_APIC
4 //block 2
5 #endif
```

Listing 2: Sample of Preprocessor Scheme

Currently the preprocessor annotation is also used in the benchmark in other programming languages, like Java [13]. Such annotation strategy will not affect the execution of the program, since they are always placed in the comments in the program.

In caIDE, we adopt a colored annotation strategy for our benchmark. We choose a colored annotation scheme based on two considerations.

- The colored annotation will provide visual support for developers. As discussed in [14], the background colors have the potential to improve program comprehension.
- The relations between abstract syntactic tree (AST) nodes and features are stored locally. Our tool can help developers operate the annotated system easily. Such as, inspecting the interactions between features, and displaying the code fragments for certain features.

B. Feature Location

The task of feature location is to map the features with their implementations in the code base. In our benchmark, we adopt the following steps for feature location.

- **STEP 1 (recording individual feature)**. For each feature collected, we first explore the app on the physical devices and define a set of scenarios, which could represent the feature. To capture the feature at runtime, we record scenarios belonging to the feature with *Method Tracer*[3] in Android Debug Monitor (ADM). A set of quadruples {name, invocation count, inclusive time, exclusive time} in execution order is returned for representing the feature. Then, we manually annotate each features in the code base with caIDE.
- **STEP 2 (expand the annotations)** With the first step, only some methods for feature are annotated in the program. Then, the feature annotation process can be automatically conducted with our approach, which is a conditional probability based feature mining approach [15]. As demonstrated in the paper[15], our feature location approach outperforms other three feature location techniques.
- **STEP 3 (manually checking and fixing)** Finally, we manually check the annotations returned by the feature mining approach and fix inappropriate annotations. Here, we conduct such check by reading the source code of app and running the app on emulator.

V. BENCHMARK

In this section, we will introduce the components of the benchmark. In general, in the benchmark, we provide the following information and data for each app.

- For each app, we provide a list of features and a feature model. The feature model describes the relations among these features.
- The feature annotation ground-truth of each project is presented as a series of `xml` file. Each `xml` file corresponds to a `Java` file and represents the annotations in the source code. Specifically, in the `xml` file, the mapping between AST nodes and features is defined. The feature location techniques can be used on the benchmark to test their performance.
- Our tool caIDE provides visualization of the benchmark. Code fragments for different features are assigned to different background colors. Colors assigned for each feature is described in the `color.json` file.

A. Target Apps

The target apps are selected from the open-source community F-Droid[4]. We collect 1,365 open source Android apps from F-Droid in total. Then we apply the following criteria to select the target apps:

---

[3]Method Tracer: https://developer.android.com/studio/profile/am-methodtrace.html
[4]F-Droid: available at https://f-droid.org

- We only preserve those apps, that have been published via Google Play, since the apps' descriptions are mainly collected from the Google Play.
- We exclude apps that are not developed in Java.
- We exclude trial apps that have less than 3 features.
- For those apps, that do not have sufficient descriptions, we also skip them as well.

By applying these selecting criteria, 198 apps are reserved. These apps cover 22 categories from Google Play. The top categories include *tools*, *productivity*, *communication* and *music & audio*. These apps' downloads ranges from 5,000 to 100,000,000. Therefore, the apps selected for our benchmark are objective.

B. Features and Feature Model

In the benchmark, we use the approaches presented in Section III to extract features and build the feature model. The feature model of the project is defined in the `feaeturemodel.afm` file in each project's root directory. In addition, the colors bind with features are stored in the `color.json` file. When developers use caIDE to update the feature model or assign/update colors for features, these two files are updated correspondingly. The number of features in each project ranges from 2 to 19. Furthermore, the cumulative distribution of the number of features in apps is shown in Figure 3. As we can observe from Figure 3, half of subjective apps have less than 6 features. This is due to some features are implemented with third-party API and some are non-functional features. Therefore, we cannot find the corresponding code fragments in the code base for these features.
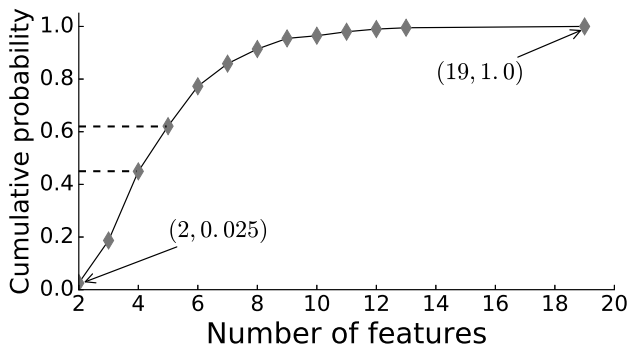

Fig. 3: The CDF of number of features in apps

By inspecting the features found from the description of the app, not all features could be considered as features for building the APL. For example, "Fast access to items" is a feature listed in *XBMC Remote* app, but it cannot be considered as a feature, since developers do not implement an algorithm for fast accessing. In addition, those non-functional features, e.g. "no ads ever", "easy to use", also cannot be mapped to code fragments. Some features' corresponding code segments are implemented with third-party APIs. For example, in a calendar app, the description contains a sentence "it can connect your Google Accounts and can be synchronized to automatically ...". In practice, developers simply use the Google API to implement this function. Here, we list several features, which are frequently implemented by third-party APIs.

- **Http Utility**: a lots of apps use the resources from website. Therefore, developers often have to check the connection status with http related libs. For example, the APIs `com.squareup.okhttp3` can be used to check the network, send the request, and diagnose.
- **Image download**: some apps provide functions to allow users download image from server. For example, the APIs in `com.squareup.picasso:picasso` provide service for image downloading and caching.
- **Material design**: some apps make the UI elements have a material look-and-feel design. For example, the APIs in `com.afollestad.material-dialogs` help developers in UI design.
- **Color picker**: some photo/image editing apps contains a module to let users directly select color from the paint. For example the lib `j4velin.colorpicker` provides such functions for users.

C. Granularity

The granularity of features in Android product lines ranges from fine to coarse. For fine granularity, a feature could be simply implemented by a *field*, a *statement*, or even a *case* block under *switch-case* structure. Whereas, some features are represented by *class*es, or even *package*s. Discarding those features, that are implemented with third-party APIs, we compute the proportions of granularity of other features used in apps. We find that the proportions of features in field/variable, statement, method and class are 24.6%, 16.5%, 60%, 53.2%, respectively.

D. Usage of Benchmark

The benchmark can be used by developers and researchers mainly for following cases.

- **Testing feature location technique proposed.** When a novel feature location technique is proposed, there is always a need for a benchmark in order to assess the performance of the feature location technique. caIDE allows developers to design their own feature location approaches and annotate the product line with customized approaches automatically.
- **Building the product variants.** caIDE allows developers to build the variant applications from the annotated product line automatically. Developers only have to set the configuration from the GUI configuration panel in caIDE. Then caIDE can extract the product variants from the annotated product with the configuration provided. We introduce the details of this function in Section VI-B.
- **Inspecting the feature implementation.** In addition, the benchmark can be used by developers to learn how to implement certain functions in the Android apps. For researchers, it can be used for software reuse study, programming practices, modularity, and other possible research issues.

VI. Supporting Functions

In this section, we will introduce several additional operations already implemented in caIDE to help developers use the benchmark easily. Besides two basic modules (annotation and view), caIDE also allow developers to analyze the interactions between features (Section VI-A), and extract product variants from the annotated product line.

A. Interactions Between Features

caIDE can also help developers analyze the interactions between features. Typically, relations and constraints between features should be described in the feature model. However, our caIDE can also help developers check the interactions and guide developers to update the feature model. For instance, if caIDE find that the execution of feature f_1 may imply the execution of feature f_2 from the annotations, it will suggest developers to add such a constraint in the feature model. Therefore, the *interaction* module in our tool not only helps developers view the interactions between features, but also tries to provide suggestions for adding additional relations in the feature model. Still, the suggestions provided by the *interaction* module for fixing the feature model are not compulsory and cannot be fully correct. This is because that the suggestions are merely collected from the code base and based on the program analysis, whereas the relations between features should be designed and confirmed only by the domain expert. The main purpose is to provide suggestions for developers with limited domain knowledge.

Moreover, we will introduce all interactions currently supported in the caIDE.

- *requires:* in an SPL, if a feature uses data from another feature, it builds a usage dependency from the data consumer to its producer. In practice, caIDE will first collect the implementations of each feature. Then, it will check whether there exists a data flow from a feature to another by traversing the program dependency graph (PDG) of the app. The PDG of an app contains both data flow dependency and control flow dependency [16].
- *mutual exclude:* the *mutual exclude* relation between two features represents that two features cannot be in the same product variant. For example, certain apps can be either in off-line mode or on-line mode. Therefore, if an app in run in an off-line mode, it always cannot in the online mode at the same time. In practice, caIDE checks such relation by detecting whether the executions of two features from two aspects: (1) whether two features are booted by different conditions; (2) whether two features are exclusive in nature from the programming perspective. For example, feature `f_CHN` is represented by a class `Chinese` and class `English` represents feature `f_GBR`. In addition, class `Chinese` and class `English` are inherited from class `Language`. Therefore, it is possible that only one feature between `f_CHN` and `f_GBR` can be existed in the variant product.

Again, all these interactions are collected based on the annotations of features. The relations and constraints for features can only be used as auxiliary information for domain experts to refine the feature model. Domain experts have to confirm these recommendations and make final decisions on their own. However, this does not mean that caIDE make the incorrect conclusions. Since caIDE makes the decision only reply on the annotations for features. The relations and constraints between features are more complex and require domain knowledge. However, caIDE does not have such domain knowledge, caIDE tries to predict the relations between features based on annotations. Therefore, caIDE can only provide hints for domain experts based on annotations.

B. Transforming the Annotation SPL to Variants

In addition, our caIDE can also help developers build the variants from the annotated systems (benchmark). In SPL, a variant represents a running system that can be used by end-users directly [3]. It is built by configuring the SPL.

Recall the annotation scheme mentioned in SectionIV-A, caIDE can help developers generate the target variants from the annotated system. caIDE adopts an AST-write strategy to build the variant. In general, it contains three steps. First, the code is parsed into an AST. Second, the AST nodes in the AST are assigned to different features as presented in the feature annotation. Then, based on the feature module of the app and the configuration given by the user, all AST nodes, that are associated with unwanted features in the configuration, are marked for removing from the AST.

VII. Discussions

Furthermore, we intend to share the experience gained and point out several handful lessons learned in this work.

Differences between desktop applications and mobile apps in terms of reuse. We compare the annotation process in mobile apps and desktop applications, we highlight the following main differences between two types of applications.

- **Call Graph vs. Window Transaction Graph.** In most desktop applications, the program starts from the `main` method, and then goes though the call graph based on the input context. The call graph describes the possible execution path of the program in a graph [17]. Whereas, in the mobile context, the window transition graph (WTG) is frequently used to analyze the execution of the app [18]. In the WTG, nodes represent windows and transitions between windows are linked with edges. Transitions are triggered by callbacks executed in the UI thread. Therefore, to reuse components in apps, developers not only have to take care of the execution logic, but also carefully resolve all UI elements and events involved.
- **Resources.** Resources (UI elements, strings and layout) are another major concern in app reuse comparing to desktop applications. Specifically, not all desktop applications have graphical user interfaces (GUI) for end users. Whereas the majority of apps have GUI provided for users. Therefore, the approach for reusing apps must carefully take care of resources. This rule is also suitable for building a software product line for Android apps.
- **Third-party API.** To successfully reuse software artifacts in apps, all third-party APIs involved in the apps

should also be inspected. Specifically, developers have to explore the functionality of those APIs, how these APIs are used, how these APIs are cooperated with others, and what parts of the program are affected. We highlight the impact of third-party APIs is because that sometimes those APIs can change some existing programming practices in Android. For example, library *Butter Knife* (`com.jakewharton:butterknife`[5]) define a new approach to manage UI elements. It is apparent that an approach for building SPL for other apps cannot be applied to the app with *Butter Knife*. Therefore, developers have to cope with these special cases separately. Unlike desktop application, it is hard to design a universal reuse approach for all apps.

- **Android API Compatibility.** Android APIs are frequently updated, which may introduce compatibility issue in reuse. For example, after API 21, Android adopts the camera2 API to use camera resource. The camera2 API follows a different pattern comparing to camera1 in terms of using camera.

Hence, researchers, who intend to propose approaches for building SPL on Android apps, have to take into account the differences between mobile apps and desktop applications. That also means that we cannot directly use the existing approaches in software reuse and software product line within the mobile context.

Extract product variants from annotation product line. In addition, we can use caIDE to extract product variants from the annotation product line (benchmark). As mentioned in Section VI-B, caIDE also provides such functions to allow developers to extract the product variants from the annotation product line automatically. By conducting such experiments, we found that even caIDE can create the variant automatically, some additional efforts are still required from developers. The additional task is mainly about fixing the UI. For example, if we remove an `activity` in a variant based on the configuration, the button, that triggers this `activity`, will become invalid. Therefore, developers have to fix the code fragments related to the button. In caIDE, we leave this part for developers based on two concerns: (1) developers may want to redesign the UI and layout; and (2) rather than fixing the UI, developers can also redesign the execution logic for the variant product and keep the UI unchanged.

VIII. CONCLUSION

In this paper, we presented caIDE, an Android Studio plugin, and a benchmark for supporting reuse software artefacts in app on feature location. The benchmark is based on several existing android apps and is designed to support research and programming practice on software reuse in the context of software product line. The benchmark can help researchers conduct reuse studies on apps rather than desktop applications. With caIDE plugin, users can directly operate on the benchmark to build the variant product or even release their own benchmarks with our tool. In the future, we intend to extend the caIDE to support more programming languages

---

[5]Butter Knife: http://jakewharton.github.io/butterknife/

used in app development, including Kolin, python, and even native C++. In addition, we will release more benchmark for other programming languages.

REFERENCES

[1] I. J. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, and A. E. Hassan, "A large-scale empirical study on software reuse in mobile apps," *IEEE Software*, vol. 31, no. 2, pp. 78–86, 2014.

[2] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, "A survey of app store analysis for software engineering," *IEEE Transactions on Software Engineering*, vol. 43, no. 9, pp. 817–847, 2017.

[3] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*, 2005.

[4] R. Abdalkareem, E. Shihab, and J. Rilling, "On code reuse from stackoverflow: An exploratory study on android apps," *Information and Software Technology*, vol. 88, pp. 148 – 158, 2017.

[5] L. Li, D. Li, T. F. Bissyand, J. Klein, Y. L. Traon, D. Lo, and L. Cavallaro, "Understanding android app piggybacking: A systematic study of malicious code grafting," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 6, pp. 1269–1284, 2017.

[6] D. Poshyvanyk, Y. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 420–432, 2007.

[7] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Bottom-up adoption of software product lines: A generic and extensible approach," in *Proceedings of the 19th International Conference on Software Product Line*, 2015, pp. 101–110.

[8] J. Martinez, T. Ziadi, T. F. Bissyand, J. Klein, and Y. l. Traon, "Automating the extraction of model-based software product lines from model variants (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 396–406.

[9] D. Batory, "Feature models, grammars, and propositional formulas," in *Software Product Lines*, H. Obbink and K. Pohl, Eds., 2005, pp. 7–20.

[10] S. She, R. Lotufo, T. Berger, A. Wäsowski, and K. Czarnecki, "Reverse engineering feature models," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 461–470.

[11] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08, 2008, pp. 311–320.

[12] J. Sincero, R. Tartler, D. Lohmann, and W. Schröder-Preikschat, "Efficient extraction and analysis of preprocessor-based variability," *SIGPLAN Not.*, vol. 46, no. 2, pp. 33–42, 2010.

[13] M. V. Couto, M. T. Valente, and E. Figueiredo, "Extracting software product lines: A case study using conditional compilation," in *2011 15th European Conference on Software Maintenance and Reengineering*, 2011, pp. 191–200.

[14] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake, "Do background colors improve program comprehension in the #ifdef hell?" *Empirical Software Engineering*, vol. 18, no. 4, pp. 699–745, 2013.

[15] Y. Tang and H. Leung, "Sticprob: A novel feature mining approach using conditional probability," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 45–55.

[16] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[17] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call graph construction in object-oriented languages," in *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '97, 1997, pp. 108–124.

[18] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "Static window transition graphs for android," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 658–668.