Identifying Features of Android Apps from Execution Traces

Qi Xin, Farnaz Behrang, Mattia Fazzini, and Alessandro Orso
Georgia Institute of Technology
Atlanta, GA, USA
{qxin6, behrang}@gatech.edu {mfazzini, orso}@cc.gatech.edu

***Abstract*—Understanding a program and the features it provides is essential for a number of software engineering tasks, including refactoring, debugging, and debloating. Unfortunately, program understanding and feature identification are also extremely challenging and time consuming activities. To support developers when they perform these activities, we propose FEATUREFINDER, an approach that aims to identify and understand the features of a program by analyzing its executions. Specifically, we defined our approach for Android apps, given their widespread use. Given an app, FEATUREFINDER generates traces that capture different properties of the app executions through instrumentation. It then leverages the user events in the trace to split the trace into segments, and clusters these segments based on their characteristics, using a classifier. Each identified cluster indicates a feature exercised in the execution. Finally, FEATUREFINDER suitably labels each identified cluster, so as to provide a human-readable description of the corresponding feature. We performed a case study in which we used FEATUREFINDER to identify features in two executions of the K-9 MAIL app. In the study, FEATUREFINDER was able to correctly identify 6 of the 11 manually identified features, which we believe is an encouraging result and motivates further research.**

***Index Terms*—Feature identification, program understanding, trace analysis**

I. INTRODUCTION

Whether developers need to debug, refactor, maintain, or generate documentation for a program, it is essential that they understand the program and the features it provides. In this paper, in particular, we are interested in the problem of identifying and understanding the features of a program by analyzing its executions. There are a number of existing approaches that instrument a program, generate execution traces that contain useful information, and analyze the generated traces [1]. To allow developers to know what exactly happened within an execution, however, these traces typically contain a great deal of low-level information (e.g., methods called and user events). Unfortunately, analyzing and understanding this kind of low-level traces is a time-consuming and complex task.

To help developers with this task, and let them better understand program executions and the features exercised therein, we propose a new technique called FEATUREFINDER. FEATUREFINDER is specifically designed to target Android apps, as they are increasingly widespread and have specific characteristics that can be leveraged when analyzing their executions. In particular, mobile apps (and Android apps in particular) are event-based and organize their features around the concepts of screens (activities) and user events.

In our context, a *feature* is a sequence of user events that exercise some functionality of an app. For example, the *login feature* of an app may consist of the following user events: clicking on the username input text box, typing a username, clicking on the password input text box, typing a password, and clicking on the login button. To identify these so-defined features for an app A, FEATUREFINDER instruments A so that, when executed, A generates a trace that contains specific runtime information. Then, for each generated trace, FEATUREFINDER identifies the user events in the trace and splits the trace into segments separated by user events. At this point, FEATUREFINDER uses a clustering algorithm to group consecutive trace segments iteratively, based on the runtime information associated with the segments. At the end of this step, each cluster represents an identified feature. FEATUREFINDER also generates a human-readable label for each cluster (i.e., feature) and outputs clusters and corresponding labels.

To evaluate FEATUREFINDER, we implemented it in a prototype tool and performed a case study in which we used the tool to identify the features in two execution traces of the K-9 MAIL app. In the study, FEATUREFINDER was able to correctly identify 55% of the features manually identified in the traces considered. These results, albeit preliminary, are promising and show that FEATUREFINDER is a potentially effective technique for identifying features in app executions. Moreover, although the current definition of FEATUREFINDER is Android-specific, the basic approach should be easily applicable to other GUI-based applications (e.g., web applications).

This paper makes the following contributions:

- An approach for identifying features of an app by analyzing its execution traces.
- An implementation of the approach for the Android platform (publicly available at https://sites.google.com/view/featureidentification).
- A case study that shows the potential usefulness of our approach.

II. APPROACH

In this section, we present FEATUREFINDER, an approach for extracting features from execution traces. Figure 1 provides a high-level overview of FEATUREFINDER. As the figure shows, the approach takes as input an app and operates in five steps: *instrumentation*, *execution*, *splitting*, *clustering*,

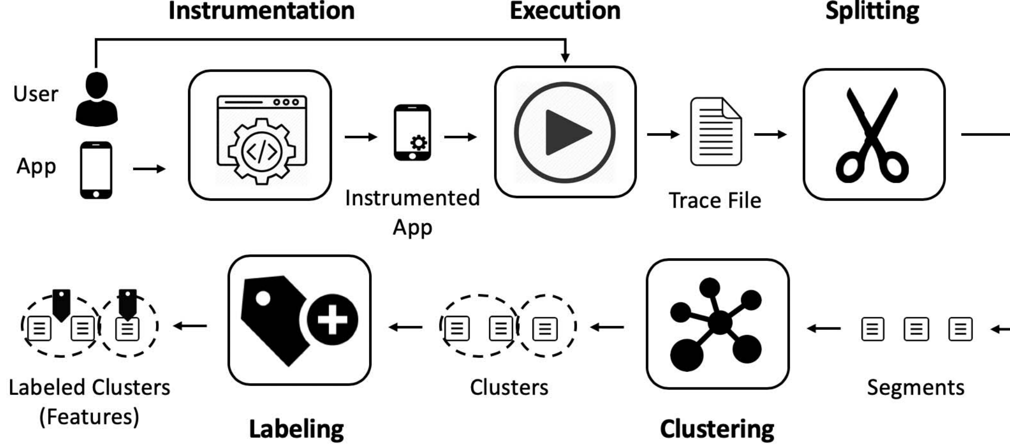

Fig. 1. High-level overview of FEATUREFINDER.

and *labeling*. The output of the approach are the features exercised by a user execution of the app. We now elaborate on FEATUREFINDER's steps.

A. Instrumentation and Execution

In the *instrumentation step*, FEATUREFINDER instruments the app to capture certain execution properties. This information is stored in an execution trace and is used in later steps to identify the set of features exercised during execution. Figure 2 illustrates, in the form of a grammar, the traces generated by FEATUREFINDER. The approach captures several execution properties: call stack at each method call, activities and fragments traversed,[1] and user events. We chose these properties as we believe they capture essential information about a user execution and the features it exercises.

Specifically, FEATUREFINDER captures call stacks (*MCStack*) to identify method calls (*MCall*) and their depth. For each method call, the approach extracts package (*PackageName*), class (*ClassName*), and method (*MethodName*) names associated with the call. It also records the names of the activities (*ActName*) and fragments (*FragName*) explored during the execution. For each activity/fragment, FEATUREFINDER records starting point (*ActStart*/*FragStart*) and ending point (*ActEnd*/*FragEnd*). FEATUREFINDER also records user events (*UserEvent*), where a user event can be either a touch event (*TouchEvent*) or a keyboard event (*KeyBoardEvent*). A touch event is associated with a widget (*Widget*) (e.g., a button), for which FEATUREFINDER records, if available, id (*WidgetId*), associated text (*WidgetText*), and content description (*WidgetContentDesc*). A keyboard event is associated with a key label (*KeyLabel*).

In the *execution step*, FEATUREFINDER uses the instrumentation to produce a trace that captures the execution properties observed while a user is interacting with the app.

B. Splitting and Clustering

FEATUREFINDER uses a bottom-up approach to identify features within a trace. Intuitively, the approach recognizes features by grouping parts of the trace together. More precisely, FEATUREFINDER recognizes features in its splitting and clustering steps.

[1]In Android apps, activities and fragments are basically screens of the app.

```
Tr -> E Tr | epsilon
E  -> { MCStack } | ActStart | FragStart | ActEnd | FragEnd
      | UserEvent
MCStack -> MCall MCStack | epsilon
MCall -> ( PackageName ClassName MethodName )
ActStart -> ActName *as*
FragStart -> FragName *fs*
ActEnd -> ActName *ae*
FragEnd -> FragName *fe*
UserEvent -> < *tevent* Widget > | < *kbevent* KeyLabel >
Widget -> WidgetId WidgetText WidgetContentDesc
```

Fig. 2. Trace grammar.

In the *splitting step*, the approach divides the trace into *segments* based on the occurrence of touch events. We use these events to determine segments because we believe that features are usually invoked through an explicit action performed by the user (e.g., clicking on the login button).

After this step, the *clustering step* groups consecutive segments that are deemed to be related. We use the term *cluster* to refer to a group of segments. The algorithm that FEATUREFINDER uses to cluster segments is shown in Algorithm 1. The algorithm follows a "snowball-growing" principle; that is, it keeps grouping the current segment with the subsequent one if the classifier directs it to *merge* (i.e., to grow the "snowball"). Specifically, the algorithm starts with an initialization step (lines #1–3) and then keeps iterating over the segments in the trace (lines #4–14). Each iteration identifies one cluster. At line #5, the algorithm creates a new cluster, which contains the first available segment. It then processes subsequent segments (lines #6–11) and tries to merge these segments into the current cluster using the classifier. The algorithm stops grouping segments when either the classifier (described below) returns a *don't merge* as an answer or the trace has been fully processed. The list of computed clusters is the output of the algorithm, where each cluster indicates a feature in the app.

The classifier used by FEATUREFINDER determines whether two segments should be clustered based on the execution

Algorithm 1 Clustering algorithm.

```
Input: uts: list of segments
       classifier: classifier to group segments
Output: clusters: list of grouped segments
 1: clusters ← {}
 2: int ut_size ← uts.size()
 3: int i ← 0
 4: while true do
 5:   int begin ← i
 6:   for i ← begin + 1; i < ut_size; i + + do
 7:     segment_0 ← MERGE(uts, begin, i − 1)
 8:     segment_1 ← uts.get(i)
 9:     label ← classifier.classify(segment_0, segment_1)
10:     if label ==“don’t merge” then
11:       break
12:   clusters.add(MERGE(uts, begin, i − 1))
13:   if i == ut_size then
14:     break
15: return clusters
```

properties contained within the segments. To do so, the classifier uses a feature vector that contains the 29 numeric features shown in Table I, which encode how two segments relate to each other. (We identified these features based on our expertise and preliminary evaluation. In future work, we plan to investigate how individual features contribute to the classification accuracy and to possibly consider additional features.) Given the feature vector for a pair of segments, the classifier predicts whether the segments should be merged (“merge”) or not (“don’t merge”).

C. Labeling

In this step, FEATUREFINDER associates to each identified cluster C a label that is meant to provide a human-readable description of the feature represented by C. To compute the label for C, the approach first collects the names of the activities and fragments present in C. It then (1) treats these names as terms, (2) computes the tf-idf value for each term considering C as a document, (3) ranks the terms based on the computed values, and associates the set of the top-10 terms to C as its label. The output of the technique is the set of labeled clusters, where each cluster represents a feature in the trace.

III. CASE STUDY

To evaluate FEATUREFINDER, we conducted a case study on five randomly selected open-source apps that belong to different categories: K-9 MAIL [4], WORDPRESS [5], DAILY-MONEY [6], PASSWORDMAKER [7], and MUSIC PLAYER [8]. We first instrumented the apps using FEATUREFINDER. Then, for each app, we (1) created two usage scenarios that exercised different features of the app and (2) obtained two traces by realizing the scenarios while running the app. Table II shows the resulting traces, which contain 60.6 user events on average.

We randomly selected to use K-9 MAIL’s traces for testing, and the eight traces of the remaining apps for training. For each of these eight traces, we used FEATUREFINDER to split the trace into segments. One of the authors then manually identified clusters for each trace by determining which segments corresponded to a feature and should have been clustered. Based on the identified clusters, we then labeled each pair of feature vectors for contiguous trace segments as either “merge” or “don’t merge”, based on whether the segments belonged to a manually identified cluster. Using this approach, we generated a total of 490 labeled pairs of trace segments and corresponding feature vectors. We used these pairs to train, using 10-fold cross-validation, 10 commonly used classifiers available in the Weka package [9], including classifiers based on decision trees, logistic regression, SVM, and k-NNs. As the results in Table III show, these classifiers can achieve high accuracy (0.84 on average).

Based on these results, we implemented a prototype of FEATUREFINDER using IBk(10), which has the highest accuracy among the classifiers considered. We then used our prototype on the two traces of K-9 MAIL, using the manually identified clusters as ground truth. (To mitigate the risk of bias, an additional author inspected and agreed on the manually identified clusters.) Table IV shows the clusters we manually identified (*ground truth*) and the labels that we created to indicate the corresponding features. The table also shows the clusters and labels computed by FEATUREFINDER. Due to space limitations, we do not show the user events, which are available elsewhere [10].

As shown in Table IV, FEATUREFINDER identified six features for Trace 0 (*TID=0*), among which four match the ground truth: (*ft01*, *fh01*), (*ft02*, *fh02*), (*ft05*, *fh05*), and (*ft06*, *fh06*). (Note that some starting and ending segment ids do not match exactly due to noise in the traces, such as a touch event on the back button.) For feature *fh04* (*Adding account*), FEATUREFINDER identified two features: *ft03* (which corresponds to providing an email address and a password to login) and *ft04* (which corresponds to setting a name for the new account). As for *fh03* (*Email setting*), which involves adding stars for two emails, FEATUREFINDER grouped it with the login part of adding an account (*ft03*), most likely due to non-significant changes with respect to the activities, fragments, and method calls in the corresponding cluster.

For Trace 1 (*TID=1*), FEATUREFINDER identified three features, among which two match the ground truth: (*ft11*, *fh11*) and (*ft13*, *fh15*). (Also in this case, segment ids do not match exactly in some cases, for the same reasons mentioned above.) Feature *ft12* groups together three ground-truth features: *fh12*, *fh13*, and *fh14*, which correspond to email composing, email search, and invoking the “About” menu entry. In this case, the recorded execution properties were not enough for FEATUREFINDER to split the sequences correctly. Overall the differences between sequences are not considered as significant by the classifier due to the presence of activities that are present in a significant number of segments (e.g., *MessageList* and *SettingsActivity*).

For the six features that FEATUREFINDER correctly identified, we believe that the labels computed by FEATUREFINDER effectively capture the essence of the features. For three of the features (*ft02*, *ft06*, and *ft13*) the labels contain words present in the ground-truth labels (e.g., *folder*), and all the labels are close in meaning to the ground-truth labels.

Overall, FEATUREFINDER correctly identified 55% of the features in the traces considered (4/6 for Trace 0 and 2/5 for Trace 1). These features provide a high-level summary

TABLE I
FEATURES USED TO COMPARE TWO TRACE SEGMENTS.

ID	Name	Description
VF0	size	0, if both small; 1, if one small and one large; 2, if both large.*
VF1	clustering	0, if both are individual segments; 1, if one of the segments is a cluster.
VF2	activity_usage	Jaccard similarity of the activities in the two segments, treated as sets.
VF3	fragment_usage	Jaccard similarity of the fragments in the two segments, treated as sets.
VF4	widget_usage	Jaccard similarity of the widget information in the two segments, treated as sets.
VF5-VF12†	package_words	Similarity of the words‡ contained in the package names occurring in the two segments.
VF13-VF20†	class_words	Similarity of the words‡ contained in the class names occurring in the two segments.
VF21-VF28†	method_words	Similarity of the words‡ contained in the method names occurring in the two segments.

* A segment is considered small if it contains less than three method calls, large otherwise.
† FEATUREFINDER computes the cosine similarity of eight pairs of vectors, where FEATUREFINDER generates the first (resp., second) vector in each pair based on the words contained in the first (resp., second) segment, and eight combinations are obtained by (1) populating the vectors with either tf or tf-idf values for the words [2], while (2) treating the words as either sets (i.e., without repetitions) or bags (i.e., with repetitions), and (3) considering either plain words or words augmented with their depth.
‡ To extract words, FEATUREFINDER (1) splits names into tokens (e.g., based on camel case), (2) performs stemming [3], (3) eliminates Java keywords, stop words, and tokens whose length is less than 3 or greater than 32, and (4) makes the remaining tokens lower case.

TABLE II
TRACES USED FOR TRAINING AND FOR TESTING.

TID	App	Category	#MethodCall	#Segment	#Vector‡
0	WORDPRESS	Productivity	16,743	74	73
1	WORDPRESS	Productivity	22,000	63	62
2	DAILY-MONEY	Finance	12,080	92	91
3	DAILY-MONEY	Finance	19,108	84	83
4	PASSWORDMAKER	Tools	2,080	23	22
5	PASSWORDMAKER	Tools	2,434	34	33
6	MUSIC PLAYER	Media	32,365	57	56
7	MUSIC PLAYER	Media	36,258	71	70
0	K-9 MAIL	Communication	23,675	58	n/a
1	K-9 MAIL	Communication	48,129	50	n/a

‡ Number of feature vectors used for the training of the classifier.

TABLE III
ACCURACY OF THE CLASSIFIERS CONSIDERED.

Weka Classifier	Accuracy	Classifier Description
J48	0.83	A decision tree classifier
PART	0.82	A decision list classifier
DecisionTable	0.84	A simple decision table majority classifier
Logistic	0.83	A multinomial logistic regression model
SMO	0.85	A support vector classifier
IBk(1)	0.83	A k-NN classifier (with k=1)
IBk(2)	0.86	A K-NN classifier (with k=2)
IBk(3)	0.83	A K-NN classifier (with k=3)
IBk(5)	0.83	A K-NN classifier (with k=5)
IBk(10)	0.86	A K-NN classifier (with k=10)

TABLE IV
FEATURES IDENTIFIED FOR K-9 MAIL.

TID	Tool/Human	Features	
		IDs	Labeled Clusters ([sid,eid]:label0,label1)‡
0	FEATURE FINDER	ft01	[0,16]:MessageList,MessageListFragment
		ft02	[17,25]:FolderSettings,FolderList
		ft03	[26,35]:SettingsActivity,AccountSetupBasics
		ft04	[36,39]:AccountSetupNames,AccountSetupCheckSettings
		ft05	[40,44]:MessageViewFragment,MessageList
		ft06	[45,49]:MessageCompose,MessageList
	Ground truth	fh01	[0,15]:Email checking
		fh02	[16,26]:Managing folders
		fh03	[27,30]:Email setting
		fh04	[31,39]:Adding account
		fh05	[40,43]:Email checking
		fh06	[44,49]:Email composing
1	FEATURE FINDER	ft11	[0,25]:MessageList,MessageListFragment
		ft12	[26,42]:AboutActivity,MessageCompose
		ft13	[43,57]:GeneralSettingsFragment,GeneralSettingsActivity
	Ground truth	fh11	[0,24]:Email checking
		fh12	[25,30]:Email composing
		fh13	[31,35]:Email search
		fh14	[36,42]:Getting info
		fh15	[43,57]:General settings

‡ Labeled clusters are shown in the form of [sid,eid]:label0,label1, where sid and eid are the starting and ending ids of the segments, and label0 and label1 are the top-2 terms in the labels (we only show the top-2 terms due to space limit).

of the execution trace and can help the developer understand the corresponding execution. FEATUREFINDER also reported some incorrect features, however, which may mislead the developer. To mitigate this issue, in the future, we plan to provide better guidance to the developer by associating confidence values to the identified features.

IV. RELATED WORK

Our technique is related to techniques that do trace analysis for program understanding. These techniques use different approaches, such as trace reduction and compression [11], [12], summarization [13], segmentation [14], pattern mining [15], and visualization [16], [17]. In particular, FEATUREFINDER is most closely related to techniques that do trace segmentation to identify execution phases, such as those that consider similarity among method calls [18], [19], deltas in call-stack depth [14], method call frequency [20], object creation and deletion [21], [22], and program structure [23]. FEATUREFINDER differs from these techniques in that it uses a classifier-based algorithm in identifying and clustering related user events for feature identification. Our technique is also related to, albeit different from, techniques for feature location (e.g., [24]) and techniques that use dynamic analysis for program understanding (e.g., [1]).

V. CONCLUSION & FUTURE WORK

We presented FEATUREFINDER, which aims to dynamically identify features of Android apps. Given an app, FEATUREFINDER first instruments it to generate traces that capture different execution properties. It then splits the trace into segments and uses a bottom-up approach to cluster consecutive, related segments, where each cluster indicates a feature. Finally, it labels each cluster to provide a human-readable description of the corresponding feature. We also presented a case study that shows the viability of our approach.

In addition to the future work we described earlier in the paper, we plan to extend FEATUREFINDER so that it can identify features hierarchically, at different levels of abstraction. We will also define a visualization for presenting the identified features to the users. Finally, we plan to extend our evaluation by including more apps and conducting a user study.

ACKNOWLEDGMENTS

This work was partially supported by NSF, under grants CCF-1161821 and 1563991, DARPA, under contracts FA8650-15-C-7556 and FA8650-16-C-7620, ONR, under contract N00014-17-1-2895, and gifts from Google, IBM Research, and Microsoft Research.

REFERENCES

[1] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Transactions on Software Engineering (TSE)*, vol. 35, no. 5, pp. 684–702, 2009.

[2] M. A. Russell, *Mining the Social Web: Analyzing Data from Facebook, Twitter, LinkedIn, and Other Social Media Sites*, 1st ed. O'Reilly Media, Inc., 2011.

[3] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[4] (2019, Jan.) K-9 mail. [Online]. Available: https://github.com/k9mail/k-9/releases/tag/5.600

[5] (2019, Jan.) Wordpress. [Online]. Available: https://github.com/wordpress-mobile/WordPress-Android

[6] (2019, Jan.) daily-money. [Online]. Available: https://github.com/dennischen/daily-money

[7] (2019, Jan.) Passwordmaker. [Online]. Available: https://github.com/passwordmaker/android-passwordmaker

[8] (2019, Jan.) Music player. [Online]. Available: https://github.com/MaxFour/Music-Player

[9] (2019, Jan.) Weka. [Online]. Available: http://weka.sourceforge.net/packageMetaData/

[10] (2019) User events of the two traces for k-9 mail. [Online]. Available: https://github.com/featurefinder/feature_identification

[11] J. Bohnet, M. Koeleman, and J. Döllner, "Visualizing massively pruned execution traces to facilitate trace exploration," in *Proceedings of 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. IEEE, 2009, pp. 57–64.

[12] S. P. Reiss and M. Renieris, "Encoding program executions," in *Proceedings of the 23rd International Conference On Software Engineering (ICSE)*. IEEE, 2001, pp. 221–230.

[13] A. Hamou-Lhadj and T. Lethbridge, "Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system," in *Proceedings of 14th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 2006, pp. 181–190.

[14] Y. Feng, K. Dreef, J. A. Jones, and A. van Deursen, "Hierarchical abstraction of execution traces for program comprehension," in *Proceedings of the 26th International Conference on Program Comprehension (ICPC)*. IEEE, 2018, pp. 86–96.

[15] S. Alimadadi, A. Mesbah, and K. Pattabiraman, "Inferring hierarchical motifs from execution traces," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, 2018, pp. 776–787.

[16] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. Van Wijk, and A. Van Deursen, "Understanding execution traces using massive sequence and circular bundle views," in *Proceedings of 15th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 2007, pp. 49–58.

[17] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang, "Visualizing the execution of java programs," in *Software Visualization*. Springer, 2002, pp. 151–162.

[18] H. Pirzadeh and A. Hamou-Lhadj, "A novel approach based on gestalt psychology for abstracting the content of large execution traces for program comprehension," in *Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2011, pp. 221–230.

[19] S. Medini, V. Arnaoudova, M. Di Penta, G. Antoniol, Y.-G. Guéhéneuc, and P. Tonella, "Scan: an approach to label and relate execution trace segments," *Journal of Software: Evolution and Process*, vol. 26, no. 11, pp. 962–995, 2014.

[20] A. Zaidman and S. Demeyer, "Managing trace data volume through a heuristical clustering process based on event execution frequency," in *Proceedings of the Eighth European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2004, pp. 329–338.

[21] Y. Watanabe, T. Ishio, and K. Inoue, "Feature-level phase detection for execution trace using object cache," in *Proceedings of the 2008 international workshop on dynamic analysis (WODA)*. ACM, 2008, pp. 8–14.

[22] O. Benomar, H. Sahraoui, and P. Poulin, "Detecting program execution phases using heuristic search," in *International Symposium on Search Based Software Engineering (ISSRE)*. Springer, 2014, pp. 16–30.

[23] T. Wang and A. Roychoudhury, "Hierarchical dynamic slicing," in *Proceedings of the 2007 international symposium on Software testing and analysis (ISSTA)*. ACM, 2007, pp. 228–238.

[24] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.