Crowdsourced Requirements Generation for Automatic Testing via Knowledge Graph

Chao Guo
Tieke He*
Wei Yuan
Yue Guo
Rui Hao
State Key Laboratory for Novel Software Technology, Nanjing University, China

ABSTRACT

Crowdsourced testing provides an effective way to deal with the problem of Android system fragmentation, as well as the application scenario diversity faced by Android testing. The generation of test requirements is a significant part of crowdsourced testing. However, manually generating crowdsourced testing requirements is tedious, which requires the issuers to have the domain knowledge of the Android application under test. To solve these problems, we have developed a tool named *KARA*, short for Knowledge Graph Aided Crowdsourced Requirements Generation for Android Testing. *KARA* first analyzes the result of automatic testing on the Android application, through which the operation sequences can be obtained. Then, the knowledge graph of the target application is constructed in a manner of pay-as-you-go. Finally, *KARA* utilizes knowledge graph and the automatic testing result to generate crowdsourced testing requirements with domain knowledge. Experiments prove that the test requirements generated by *KARA* are well understandable, and *KARA* can improve the quality of crowdsourced testing. The demo video can be found at https://youtu.be/kE-dOiekWWM.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Android GUI Testing, Crowdsourced Requirements, Knowledge Graph

ACM Reference Format:
Chao Guo, Tieke He, Wei Yuan, Yue Guo, and Rui Hao. 2020. Crowdsourced Requirements Generation for Automatic Testing via Knowledge Graph. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3395363.3404363

*Corresponding author: hetieke@gmail.com

1 INTRODUCTION

Android applications occupy most of the mobile application market. Due to the diversity of Android platform brands and the Android system versions, the fragmentation of the Android system is becoming increasingly severe. Various application scenarios and frequent iteration cycles make Android testing difficult. More and more developers solve the dilemma facing Android testing by using automatic testing and crowdsourced testing.

Android automatic testing is a test method which traverses the Android page components and explores the application state by simulating user interaction events and system events. It can detect potential errors or exceptions in the application by generating testing inputs [5]. The errors and exceptions can be called suspicious bugs because it is unclear whether suspicious bugs during the automatic testing process are real bugs. Moreover, the results of Android automatic testing are difficult for programmers or testers to understand. So in most of the cases, it's hard for programmers and testers to acquire effective crowdsourced testing requirements from the test results directly.

Crowdsourced testing refers to the requesters initiate test tasks to workers who are from different regions. The workers complete the tasks and submit reports [2]. Traditionally, the crowdsourced testing task includes the target application and test requirements. The crowd workers follow the instruction in the test requirement to complete the test task. However, manually generating test requirements is tedious for requesters. It requires requesters to gather domain knowledge of the target application.

In order to resolve the problem of requirements generation, we take advantage of the knowledge graph to automatically obtain the domain knowledge for the target application. As a result, we have developed the *KARA* to generate crowdsourced testing requirements automatically. *KARA* first builds a GUI model and a knowledge graph with domain knowledge for the target application. Then, the search algorithm is adopted to obtain the shortest operating path from the application entry interface to the suspicious bug according to the GUI model. At last, *KARA* utilizes the knowledge graph and the shortest operating path to generate test requirements.

In this paper, we mainly make the following contributions.

- We propose the *KARA* tool, which generates crowdsourced testing requirements for Android application automatically.
- We are the first to come up with taking advantage of the knowledge graph to assist in generating descriptions of crowdsourced testing requirements.

2 RELATED WORK

Crowdsourced testing for mobile applications is receiving increasing attention. Komarov et al. [4] compared crowdsourced GUI testing with traditional laboratory testing and proved the feasibility of crowdsourced GUI testing. Maria et al. [5] collected application data from the operations of crowdsourced workers on the application, and reproduced the context scenarios of the exception to help developers locate the cause of the exception.

Automated testing is a process that translates human-driven testing behaviours into machine execution. Amalfitano et al. [1] proposed a GUI ripping-based Android automated testing technology AndroidRipper, which uses a depth-first traversal strategy to explore different states of the program operation. It can explore the GUI changes from different startup states with manual assistance. Hao et al. [3] implemented a universal UI automation framework PUMA, which has certain extensibility, allowing testers to generate application state models based on different exploration strategies.

With the advent of sophisticated Natural Language Processing algorithms, knowledge graph has become popular. Anmol Nayak et al. [6] proposed a knowledge graph creation tool , which can extract the test intent from a requirement statement. It can be used for automated generation of test cases from natural (domain) language requirement statements.

3 IMPLEMENTATION

The architecture of *KARA* is shown in Figure 1. *KARA* first analyzes the results of the Android automated testing on the target application to obtain window transitions. Then, *KARA* constructs the GUI model for the target application. The Dijkstra algorithm is adopted to calculate the shortest operation sequence from the application starting interface to the suspicious bug interface. In the meanwhile, the static analysis technology of lexical analysis is utilized to get widget information from the source code. Based on the result of lexical analysis and transition information of the GUI model, *KARA* constructs a knowledge graph. Ultimately, *KARA* combines the shortest operation sequence with the knowledge graph to generate crowdsourced testing requirements so that crowdsourced workers can reproduce the suspicious bugs of test requirements.

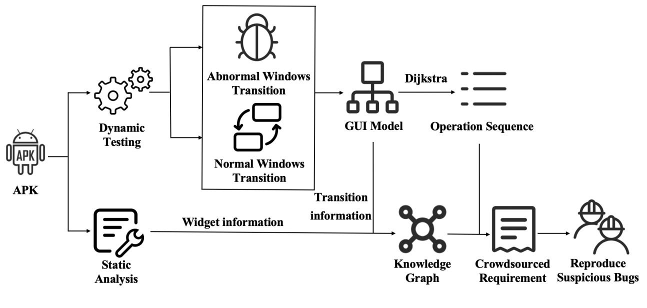


Figure 1: The Architecture of *KARA*

3.1 Operation Sequence Acquisition

In this paper, we use the Android automatic testing tool of Mooctest platform[1], which is assisted by manual test scripts. It uses the input parameters in the manual test scripts when the input widget is encountered during traversal, which can obtain comprehensive and sufficient test results. According to the test results of the Android automatic testing on the target application, *KARA* can get the test execution events. *KARA* constructs the GUI model based on normal and abnormal window transitions information, which is captured during the execution of the events. The Dijkstra algorithm is used to acquire the shortest sequence of events from the application entry interface to the triggered abnormal window transitions interface.

3.1.1 Normal Windows Transition. To traverse the events in automated testing process, we use a directed graph G=<V, E> to represent the state changes of the GUI. Vertice set V=<$V_1, V_2, ..., V_n$> is an application window list, and edge set E=<$E_1, E_2, ..., E_n$> is a window transition list. For E_i=<$V_s, V_t, m, t, c, b, P_e$>, V_s and V_t represent the starting window and the target window respectively. The operation information of triggering the window transition is recorded in m. t represents the time of the event triggered. The classification of window transition event is c. The default value 1 represents the normal window transition event, 2 represents the abnormal window transition event. The detail of the abnormal event information is recorded in b. Before the event is triggered, a screenshot would be stored in P_e. There may be multiple transitions between two windows. Various edges can exist in the same direction between V_s and V_t. However, the operation information m of E_x are different.

In the process of automated testing, the Android automated testing tool would save screenshots of the Android activity. The screenshot list can be represented as P=<$P_1, P_2, ..., P_n$>. For any screenshot P_i=<n, t>, its file name is n, and the timestamp is t. In the set of E, there must exist an edge E_j whose t_j is greater than and closest to t. Then, P_i is stored in P_e of E_j. The code is shown in Algorithm 1 (lines 2-8). *KARA* establishs a mapping relationship between screenshots and window transitions. It uses screenshots to assist in displaying test steps for crowdsourced requirements.

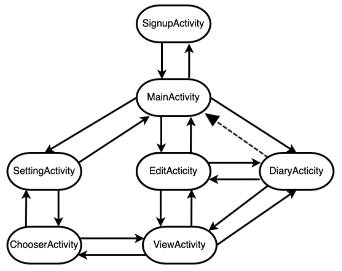


Figure 2: GUI model of the application "JianShi"

3.1.2 Abnormal Windows Transition. *KARA* utilizes the results of automatic testing to establish exception logs and window transitions mapping. The exception log set is L=<$L_1, L_2, ..., L_n$>. For any exception log L_i=<m, t>, the timestamp of log formation is t. In set of E, there must exist an edge E_k whose t_k is smaller than and closest to the t, thus we can establish the mapping between L_i and E_k. The information m of L_i is stored in b_k of E_k, and c_k is set to 2 (lines 10-16).

Part of the GUI model of Android application "JianShi" is shown in Figure 2. For the convenience of illustration, An edge is selected in the direction when various edges exist between two nodes. The

[1]http://www.mooctest.net/

solid arrow in the Figure 2 represents the normal window transition, and the dotted arrow represents the abnormal window transition.

3.1.3 Get the Shortest Operation Sequence. From the GUI model, the operation sequence S=<$E_1, E_2, ..., E_n$> of the application entry interface reaching the abnormal window transition E_n can be obtained. The *bfs* is used to get operation sequence S_t=<$E_1^{'}, E_2^{'}, \ldots, E_m$>, which is the shortest sequence from the application entry interface to the third-to-last event E_m of S (line 19-20). Then the last two events E_{n-1}, E_n are added to S_t. Then *KARA* forms the shortest operation sequence $S^{'}$=<$E_1^{'}, E_2^{'}, ..., E_m, E_{n-1}, E_n$> (line 21). Experience shows that the last three steps in the operation sequence can guarantee the correctness of the scene. If the event length of S is less than three, the shortest operation sequence of the application entry interface to E_n is obtained directly through the *bfs* (line 24).

Algorithm 1

```
Input: $G = (V, E), P, L$
Output: $S^{'}$
1:  Procedure mappingScreenshot(EventSet $E$, ScreenshotSet $P$):
2:  for all $Pi \in P$ do
3:      for all $E_j \in E$ do
4:          if $E_j.t > P_i.t$ and $E_j.t <= P_{i+1}.t$ then
5:              $E_j.P_e \leftarrow P_i$
6:          end if
7:      end for
8:  end for
9:  Procedure mappingBugLog(EventSet $E$, bugLogSet $L$):
10: for all $L_i \in L$ do
11:     for all $E_k \in E$ do
12:         if $E_k.t > L_{i-1}.t$ and $E_k.t <= L_i.t$ then
13:             $E_k.b \leftarrow L_i, E_k.p \leftarrow 2$
14:         end if
15:     end for
16: end for
17: Procedure getShortestPath(SequenceList $S$):
18: if $S.size() > 3$ then
19:     $S_t$=<$E_1^{'}, E_2^{'}, ..., E_m$>
20:     $S^{'} \leftarrow bfs(S_t)$
21:     $S^{'}.add(E_{n-1}), S^{'}.add(E_n)$
22:     return $S^{'}$
23: else
24:     return $bfs(S)$
25: end if
```

3.2 Knowledge Graph Construction

In the traditional crowdsourced testing, the generation of test requirements requires the requesters gathering domain knowledge of the target application. Fortunately, the knowledge graph can solve the issue of containing domain knowledge. It includes entities, relationships between entities and attributes to represent the ontology. In this paper, the data of knowledge graph construction comes from two parts: the GUI model and the source code of the target application. Based on the GUI model, the transition information can be acquired to form part of the entities and the corresponding relationship in the knowledge graph. The attributes in the knowledge graph can be composed of the attribute information of the widgets from the source code. Therefore, we utilize the GUI model to construct the skeleton structure of a knowledge graph and then use source code to enrich the contents of entities and attributes.

We define the type of entities in two different perspectives. From the interaction view, there are three kind of entities: 1) **Head Entity**: The entity that appeared before an action is performed. 2) **Tail Entity**: The entity that appears after an action is executed. 3) **Action Entity**: The entity only acts as an action. As shown in Figure 3, The Click Action is executed on **MainActivity**, then the interface is jumped to **EditActivity**. Thus, **MainActivity** is a Head Entity, the Click Action is an Action Enity and **EditActivity** belongs to Tail Entity. From the widget affiliation perspective, another two entity types are defined: 1) **Master Entity**: An entity that owns other entities. 2) **Slave Entity**: An entity that belongs to a certain Master Entity. In the Figure 3, that a button as Slave Entity belongs to **MainActivity** as Master Entity means a button is in the **MainActivity** interface. In our knowledge graph ontology, the relationship between entities can be ***BelongTo***, ***Take*** or ***Result***. The ***BelongTo*** illustrates the relationship between widgets and layouts. The ***Take*** and ***Result*** will delineate the cause and effect of actions. Attributes describe the characteristics of entities in a certain situation, which can be defined as the following four types: 1) **ID**: the unique identifier of the entity. 2) **Description**: the description of the operation. 3) **Type**: the value is either Click, Input or Slide. Action Entity node, Master Entity node, Slave Entity node have this attribute. An example of the Knowledge Graph body structure is shown in Figure 3.

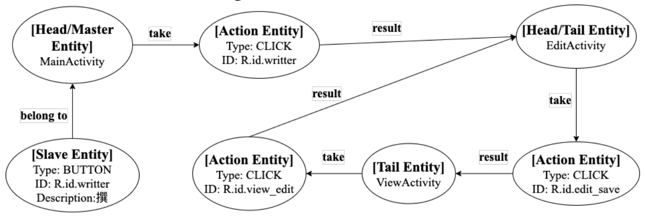


Figure 3: An Example of Knowledge Graph of *KARA*

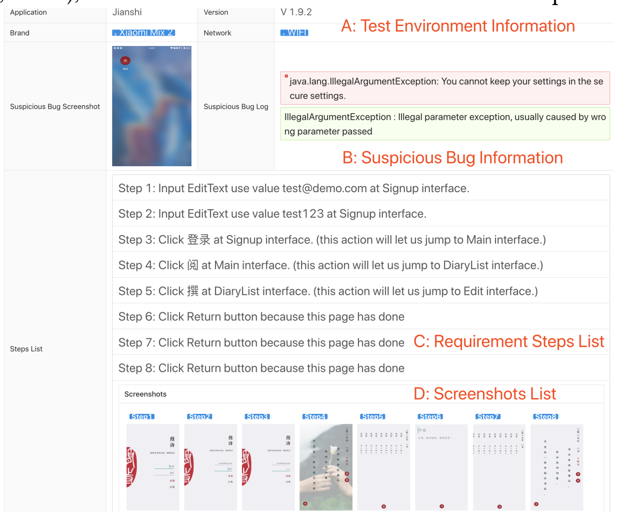
3.3 Requirement Generation

From the GUI model, *KARA* can get the shortest operation sequence of actions (like the following code), while it cannot acquire the detailed information of widget, resulted in less informative requirements generation.

```
1.Input android.jianshi:id/email use vaule test@demo.
2.Input android.jianshi:id/password use vaule test123.
3.Click widget android.jianshi:id/login.
4.Click widget android.jianshi:id/reader.
5.Click widget android.jianshi:id/view_write.
6.Click Return button because this page has done.
7.Click Return button because this page has done.
8.Click Return button because this page has done.
```

The knowledge graph is the roadmap of interface interaction meanwhile contains the knowledge of each widget. Based on the shortest operation sequence and the knowledge graph, *KARA* can generate the crowdsourced requirements. Each event E of the shortest operation sequence $S^{'}$ contains event information m. According to the widget information in m, the corresponding widget attribute can be found in the knowledge graph. Then, *KARA* optimizes the widget information description and the window transition information for m from the knowledge graph. The requirement steps after using the knowledge graph are depicted in Figure 4(PartC).

An example of crowdsourced testing requirements is shown in Figure 4. The requirement includes four parts: Test environment information (Figure 4, PartA), which includes the information of application name, application version, device brand and network condition. Suspicious bug information (Figure 4, PartB), which consists of the log information and screenshot of the suspicious bug. Requirement steps list (Figure 4, PartC), which contains the operation description of each step. It clearly defines interface transitions and hints for the next transition interface. Screenshots list (Figure 4, PartD), which includes a series of screenshots of the step.


A: Test Environment Information
B: Suspicious Bug Information
C: Requirement Steps List
D: Screenshots List
Screenshots
Step1
Step2
Step3
Step4
Step5
Step6
Step7
Step8

Application	Jianshi	Version	V 1.9.2
Brand	Xiaomi Mix 2	Network	WIFI
Suspicious Bug Screenshot		Suspicious Bug Log	java.lang.IllegalArgumentException: You cannot keep your settings in the secure settings. IllegalArgumentException : Illegal parameter exception, usually caused by wrong parameter passed

Steps List

Step 1: Input EditText use value test@demo.com at Signup interface.
Step 2: Input EditText use value test123 at Signup interface.
Step 3: Click 登录 at Signup interface. (this action will let us jump to Main interface.)
Step 4: Click 阅 at Main interface. (this action will let us jump to DiaryList interface.)
Step 5: Click 撰 at DiaryList interface. (this action will let us jump to Edit interface.)
Step 6: Click Return button because this page has done
Step 7: Click Return button because this page has done
Step 8: Click Return button because this page has done

Figure 4: An example of crowdsourced testing requirement

4 USER STUDY

We conducted experiments to address the following research questions.

RQ1: How understandable are the test requirements generated by *KARA*?

RQ2: Whether the test requirements generated by *KARA* effectively improve the quality of crowdsourced testing?

To address **RQ1**, we designed a sample and conducted a survey to collect the satisfaction of twenty participants on the comprehensibility of the test requirements generated by *KARA*. The twenty participants are graduate students majoring in software engineering and have been registered on the Mooctest platform for more than one year. From the statistical outcomes, we found 90% of participants had a positive attitude towards the understandability of the test requirements. The rest had shown negative attitudes because they thought the description of suspicious bug was insufficient, which affected the judgment on the suspicious bug.

To address **RQ2**, we conducted a controlled experiment. Ten domestic and foreign Android applications were selected from the open-source website, such as *JianShi, QQplayer* etc. They have a large number of users. Ten Android devices with high occupancy in the domestic Android market were chosen, such as *HuaWei P30, Xiaomi 10Pro* etc. We had installed all the applications before the experiment. Twenty experimenters were divided into two groups whose member tested the applications on one device. Group A is a control group that test by themselves. Group B is an experimental group that test applications according to the test requirements generated by *KARA*. The bug report that contains the bug screenshot and a short description submitted by the experimenters were collected during two hours testing process. After the experiment, the professional testers checked and analyzed the bug reports.

Table 1 shows the evaluation result. B_c represents the count of bug reports submission in the two hours. B_e represents the count of valid bugs in the submission. If the applications appear unexpected results such as confusing layouts or unresponsive, we think they are valid bugs. Within two hours, group A and group B submitted 118 and 115 bug reports, of which 68 and 93 bug reports were valid bugs. The proportion of valid bugs reports were 57.6% and 80.9% in total submission bugs, respectively. The result shows that group B testing requirements generated by *KARA* submit fewer bug reports, but it has a high rate with valid bug reports. Therefore, we can found that the test requirements generated by *KARA* can effectively improve the quality of crowdsourced testing.

Table 1: The evaluation result of test requirements

App	Version	Type	Group A B_e/B_c	Group B B_e/B_c
JianShi	2.2.0	Read	7/13	9/11
TesterHome	1.0	Community	9/20	12/15
Leafpic	0.6	Tool	3/8	8/10
RGB Tool	1.4.4	Tool	8/10	9/11
SeeWeather	2.3.1	Tool	6/9	7/9
IThouse	1.0.1	Community	5/13	8/10
QQplayer	3.2	Player	2/6	3/4
GuDong	1.8.0	Sport	5/11	7/11
AnkiDroid	2.8.4	Study	12/15	13/14
SolarCompass	1.0	Location	11/13	17/20

5 CONCLUSION

We developed the *KARA* to solve the problem of crowdsourced testing requirements generation. It is tedious to generate requirements manually during the process of crowdsourced testing. *KARA* uses the knowledge graph to enrich the description of the steps in the requirements. It can generate crowdsourced testing requirements automatically. Testers can obtain comprehensive requirements from *KARA*. Moreover, crowdsourced workers can reproduce suspicious bugs easily according to the requirements that generated by *KARA*.

ACKNOWLEDGEMENTS

This work was partially funded by the National Key Research and Development Program of China (2018YFB1403401) and the Fundamental Research Funds for the Central Universities (14380023).

REFERENCES

[1] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. 2012. Using GUI ripping for automated testing of Android applications. In *ASE*. ACM, 258–261.
[2] Yang Feng, Zhenyu Chen, James A Jones, Chunrong Fang, and Baowen Xu. 2015. Test report prioritization to assist crowdsourced testing.. In *ESEC/SIGSOFT FSE*. 225–236.
[3] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. 2014. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *MobiSys*. ACM, 204–217.
[4] Steven Komarov, Katharina Reinecke, and Krzysztof Z Gajos. 2013. Crowdsourcing performance evaluations of user interfaces. In *SigCHI*. 207–216.
[5] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *ESEC/SIGSOFT FSE*. ACM, 224–234.
[6] Anmol Nayak, Vaibhav Kesri, and Rahul Kumar Dubey. 2020. Knowledge Graph based Automated Generation of Test Cases in Software Engineering. In *Proceedings of the 7th ACM IKDD CoDS and 25th COMAD*. 289–295.