S-CAT: Summarization for Crowdsourced Android Testing

1st Author
School of Electrical and
Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
Email: http://www.michaelshell.org/contact.html

2nd Author
Twentieth Century Fox
Springfield, USA
Email: homer@thesimpsons.com

3rd Author
Starfleet Academy
San Francisco, California 96678–2391
Telephone: (800) 555–1212
Fax: (888) 555–1212

***Abstract*—Testing a new Android application is hard, since the android platform has a lot of portability and compatibility issues. These issues are caused by its fragmentation and diversified operating-system versions, as well as varied hardware platforms, which can hardly be addressed. Due to this fact, crowdsourcing emerged as an ideal way to perform the android testing, as workers possess devices that could cover a large amount of both the operation-system versions and the hardware platforms. Additionally, the diversified network settings of crowdsourced workers contribute to this task. However, the manual reporting process can be cumbersome and still need some professional background, which dampens the enthusiasm of the workers. In this paper, we propose a framework to Summarize the Crowdsourced Android Testing (*S-CAT*), which endeavours to ease the workload of workers during the process of crowdsourcing. Specifically, we extract activities from workers and bug patterns from a generated event sequence, and develop a *T*esting *P*rocess *D*escription (*TPD*) system to produce a textual recounting based on the extracted information. We also propose possible experimental designs to evaluate the efficacy of *S-CAT*, and present results of a pilot evaluation of our initial framework. Our results demonstrate that *S-CAT* can both effectively improves the efficiency of workers and facilitates debugging.**

I. Introduction

Recent years have witnessed explosive increase of adoption of mobile computing, which is almost ubiquitous with over 2.7 billion active mobile users according to a survey in 2014 [8]. In the meantime, an increased demand for high-quality, robust mobile applications is being driven by this growing user base that performs a series of computing tasks on "smart" devices, including daily news alert services, traffic services, restaurant and entertainment listings, and mobile commerce applications. Software testing is known to be generally expensive [34]. And mobile application testing can be more challenging due to some unique hardware and software features of mobile devices, e.g., mobile context, connectivity, small screen size, and restrictive data entry methods [37]. *Mobile context* typically includes the location, identities of nearby people, objects, as well as environmental elements. *Connectivity* issue is about the various network conditions that shall be taken into consideration in mobile application testing [21]. Due to the physical constraints of mobile devices, especially the *small screen size*, some direct presentation of web pages in mobile applications can be aesthetically unpleasant and even illegible [4]. Small buttons and labels limit users' effectiveness and efficiency in entering data, which may reduce the input speed and increase errors, thus the *restrictive data entry methods* is another critical issue in mobile application testing.

Currently, most mobile application testing is conducted through controlled laboratory experiments, i.e., human participants are required to accomplish specific tasks in a controlled laboratory setting. However, this testing manner may miss the testing of mobile context feature or other features we mentioned above. In addition to that, many mobile applications are developed by individual developer, who lacks resources of performing software testing.

Crowdsourcing refers to outsourcing works, which are traditionally performed by an employee, to an "undefined, generally large group of people in the form of an open call" [35]. It is based on a simple but powerful concept: Virtually anyone has the potential to plug in valuable information. To this point, it's natural for us to turn to crowdsourcing in performing the mobile application testing, considering all the unique features mentioned above. As groups are under various network conditions, possessing varied brands of mobile devices, using different versions of operating system releases.

However, we meet several challenges when launching the Crowdsourcing task. The first challenge is, the accuracy of data isn't unfounded, but it's impossible to vet everyone, and to restrict who posts data online defeats the entire purpose of crowdsourcing. For another, publicizing a crowdsourcing platform and establish a network of volunteers can be extremely challenging [12]. In a word, how to measure and manage quality in crowdsourcing systems has become an open problem. Currently, there are mainly two sets of approaches that aim at the quality control in crowdsourcing [2], i.e., the design-time approaches, e.g., effective task preparation and worker selection [7], [16], [27], and the other is runtime approaches, e.g., expert review, ground truth, majority consensus, contributor evaluation, real-time support and workflow management [7], [17], [18].

Android is the dominant mobile application platform, and in our initial work we mainly focus on the android testing. In our implementation of crowdsourced android testing, we

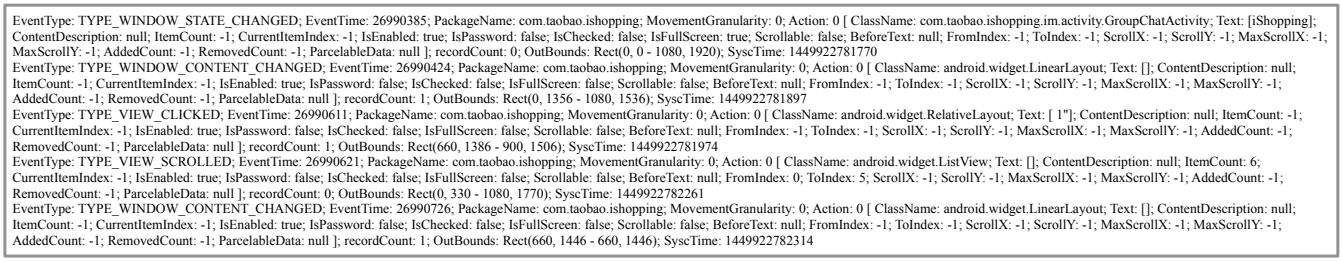

Fig. 1: An example of the event sequence

developed the Kikbug[1] platform. The working process is as follows, a requester first publishes a testing job, and workers take the job and submit the testing results before the deadline. However, during our initial deployment, we find an critical issue, which is, the workers have to manually write the bug report, for those that are not professionals, it could be complicated, also, to type in the textual description through mobile devices is somehow inconvenient, which dampens workers' enthusiasm. So we are wondering how to adopt the runtime approaches to aid the workers, and we come up with the idea of real-time support, that is, we summarize the event sequence of a worker, and then auto-generate the bug report.

AccessibilityServices is an Android framework to provide alternative navigation feedback to the user on behalf of applications installed on Android devices. Our tool takes advantage of such service to capture the working process of workers. Figure 1 shows a worker's snipped testing process. Obviously, it is not easy for humans to get intuitive view of the worker's testing processing, in terms of expressiveness and data volume. So we seek to reduce such volume, and turn to summarization. In our vision, a good textual summary of such testing process will, on one hand, ease the workers' workload, on the other hand, help the developer obtain the maximal information from the testing result of a worker, without having to inspect through the whole event sequence. A single summary could for example describe a bunch of similar testing process of different workers, or a summary could describe why a specific testing process leads to the reproduction of a certain bug. When performing crowdsourced mobile application testing, such summary can also enhance workers' experience, as they no longer have to write the bug report themselves.

The problem we target in this paper is that manually reporting during the crowdsourced android testing is cumbersome and may need some professional background, which dampens the enthusiasm of the workers. Given a bunch of evidence from various modalities collected through our Kikbug tool, including system exchange information, event sequence and screenshots, we summarize the testing results of a worker. Specifically, the *S-CAT* system first learns a classification model which decides whether a worker finds bug or not, and then it chooses the modules that are potentially bugs-related, thirdly, the system picks out the snippets of event sequence which are most probably leading to the reproduction of the bugs, lastly, a Template-based Testing Process Description (*TPD*) system produces recounting passages for the crowdsourced worker, in the format of natural language. A topic-independent planner module creates the summary by suitably concatenating the output of multiple, specialized *TPD* modules, which use a unique, manually written natural language template, respectively, to generate a sentence about event sequence, and its importance.

[1]http://kikbug.net

Bugs found? Yes

Potential Modules: login, contacts, content-share

Key testing processes:
1. In the login interface, the user first clicked the register button, and then typed in account info. in the registration interface.
2. In the account info. interface, the user scrolled down the friends list.

Fig. 2: An example of summarization

To sum up, our paper makes the following major contributions:

- We identify the challenges in mobile application testing, as well as the challenges in crowdsourcing. And come up with the idea to summarize the crowdsourced android testing, as a means of real-time support in crowdsourcing.
- We propose the *S-CAT* framework to tackle several challenges of our proposals including validating the testing process, module probability, event sequence selection, and describing the testing process.
- We conduct an extensive set of experiments based on 3 real-world android applications, and the results empirically demonstrated that the generated summary both facilitate the workers of working experience and the developers in debugging.

The reminder of this paper is organized as follows. Section II introduces the preliminary concepts and the problem statement. The *S-CAT* framework is presented in Section III. Section IV presents the design of evaluation approaches, as well as the metrics adopted. The experimental results are described in Section V, along with the discussion. Related work is presented in Section VI. Section VII concludes this article.

II. Problem Statement

In this section, we introduce some preliminary concepts, and formally define the problem of summarization for crowdsourced android testing. Table I summarizes the major notations used in the rest of the paper.

TABLE I: Notations used in the paper

SYMBOL	DEFINITION
u	crowdsourced worker
p	android application
W_p^u	working profile related to u, p
l_p^u	logcat file related to u, p
e_p^u	event sequence related to u, p
s_p^u	screenshots related to u, p
d_p^u	device info. related to u, p
a_p^u	u's activities representation of application p
$a_p^u[i]$	ith activity of a_p^u
K	the length of event sequence
L	the number of attributes
N	the number of activities

A. Preliminary Concepts

Definition 1: (Working Profile) For worker u's testing process on application p, we create a working profile W_p^u, which is a set of various modalities associated with p and u, collected via our Kikbug crowdsourcing client.

Normally, the raw working profiles consist of four aspects, however, in the real-world crowdsourced testing scenario, we observe that, the event sequence is not directly usable for summarization purpose, due to the following four reasons: (1) There are many redundant snippets of event sequences, which cause the data volume to be huge. (2) Many snippets are meaningless, resulted by meaningless actions of workers, e.g., starting an application and exiting an application. (3) For some certain events, there are attributes that are meaningless. (4) We notice that for some working profile, there are certain attributes with null value throughout the event sequence.

Therefore, in our framework, we preprocess the raw working profiles in order to make them applicable. Firstly, we remove the meaningless snippets that correspond to meaningless actions of workers. (2) Then we remove the meaningless attributes attached to certain events, for example, the attribute *ispassword* is meaningless for most events. (3) Lastly, we remove the attributes that have null value throughout the event sequence in a working profile. Details of the techniques of preprocessing raw working profiles are described in Section III.

Definition 2: (Logcat) A logcat l_p^u collects the system debug output, such as stack traces when an error is thrown and messages generated by the android *log* class.

Definition 3: (Event Sequence) An event sequence e_p^u is information obtained through the android *AccessibilityService*[2] during the working process of a testing job. It contains the UI event type, class name of the object in which the event occurred or originated, and string value representing some associated data.

Definition 4: (Device Inoformation) Device information d_p^u represents the running environment of the android application, including the brand of the device, the screen size, the resolution, the android system version, as well as the release.

```
[ 12-13 23:47:48.951  5185: 5229 D/com.netease.cloudmusic.utils.u ]
158ms on url: http://music.163.com/eapi/resource/commentInfo/list?resourceIds
=[25942063]&resourceType=4&fixliked=true

[ 12-13 23:47:49.551  5185: 5186 D/dalvikvm ]
GC_CONCURRENT freed 2132K, 13% free 35051K/40024K, paused 53ms+3ms, total 285ms

[ 12-13 23:47:49.551  5185: 5389 D/dalvikvm ]
WAIT_FOR_CONCURRENT_GC blocked 114ms

[ 12-13 23:47:49.559  5185: 5383 D/dalvikvm ]
WAIT_FOR_CONCURRENT_GC blocked 99ms

[ 12-13 23:47:49.927  5185: 5270 W/System.err ]
android.content.pm.PackageManager$NameNotFoundException: com.huawei.android.thememanager
```

Fig. 3: An example of logcat

Figure 3 shows an example of the logcat. Every record is in the $(time, processID, threadID, priority/tag, message)$ format, where the priority is one of the following character values, ordered from lowest to highest: *V - Verbose (lowest priority)*, *D - Debug*, *I - Info*, *W - Warning*, *E - Error*, *F - Fatal*, *S - Silent (highest priority)*.

Figure 1 demonstrates an example of a snippet of an event sequence. Each record represents an event, e.g., the first record stands for an event that relates to the state changing of a window type. Naturally, the receipt of an event and its associated data and its ability of querying window content present a security risk, and to mitigate this risk, in our implementation we require that *AccessibilityServices* enabled manually by the user and displays a dialog window alerting the user to the risk.

Notably, in our scenario of summarization, we may not use all the above information, but in order to best illustrate the workflow and specify the files generated along our crowdsourced android testing, we introduce all these concepts.

B. Problem Definition

In adoption of the concept "real-time support" in runtime approaches that aim at quality control in crowdsourcing [2], we come up with the idea of summarization in our specified scenario, i.e., the crowdsourced android testing. Ideally, a good bug report of a user shall contain components such as whether a bug is detected or not, the module which the bug is related, as well as the scene in which the bug can be reproduced, in the meantime, the testing environment like the hardware and operating system could be included as well. We formulate our problem as follows:

Problem 1: **(Summarization of Crowdsourced Android Testing)** Given a working profile W_p^u, our goal is to summarize the working process of worker in android testing. The output consists of three components, 1) whether bugs are found or not by the crowdsourced worker. 2) the modules that are potentially bugs-related. 3) the event sequence snippet that probably reproduces the bug. And this information is presented as well structured English readable sentences.

[2]http://developer.android.com/guide/topics/ui/accessibility/index.html

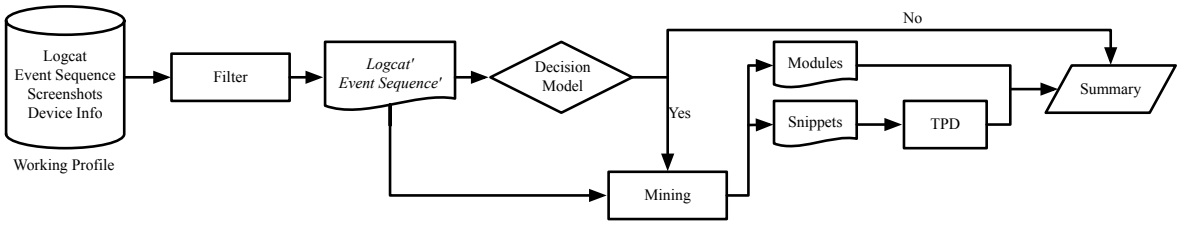

Fig. 4: Overview of *S-CAT*

III. THE S-CAT

This section describes the details of the *S-CAT* framework. Generally speaking, our approach creates a summary of the working process of a crowdsourced worker performing the android application testing. And the workflow of *S-CAT* corresponds to four major phases, as will be described in the following. The architecture of our approach is shown in Figure 4.

A. Preprocess Phase

The *Preprocess Phase* corresponds to the filtering component in the *S-CAT* architecture, which deals with the working profile. The output of this phase is the filtered *Logcat'* and *Event Sequence'*.

1) Logcat Filtering: We take advantage of the android logging system which provides the mechanism for collecting and viewing system debug output. Logs from various applications and portions of the system are collected to form the logcat file. And $(time, processID, threadID, priority/tag, message)$ is the style our customized logcat file follows, as shown in the example in Figure 3. However, the collected logcat file is large in volume, which is beyond manageable. Our goal of logcat filtering is to remove the irrelevant records which is on the kernel-level, while retain the mobile application related ones.

As mentioned in *Definition* 2, each log message has a *tag* and a *priority* associated with it. The *tag* is a short string indicating the module/component from which the message originates, and the *priority* that indicates different levels. We define a screenshot event in the `Debug` priority, which saves the screen when the worker think he/she finds a bug (but this is optional for the worker). In our initial trial of summarization, we only use the logcat file as labels, that is, we use the timestamp to locate certain activities from the event sequence, e.g., `Debug`, `Warning` and `Error`, as to identify the high risk modules or components that are potentially bugs-related. And in the meantime, we use these timestamps as evidences to extract the event sequence snippets that might reproduce the bugs. In detail, we retain log messages with the following three priorities, which is `Debug`, `Warning` and `Error`, and remove others.

2) Event Sequence Filtering: The goal of event sequence filtering is to filter out the noisy contents such as meaningless attributes and redundant snippets. Android provides a `Java API` to its accessibility framework to enable developers integrate accessibility functionality into customized applications. All drawable elements derive from a common ancestor, the View class, which contains built-in calls to *Accessibility* API methods. And in that way, most user interface widgets in the Android framework make their accessibility information available, such as the UI event type, class name of the object in which the event occurred or originated, and string value representing the associated data could be populated into an *AccessibilityEvent* object and dispatched to enabled *AccessibilityServices* via the appropriate method call. Totally, there are 25 *AccessibilityEvent* types[3].

Our collected event sequence (*AccessibilityEvent*) covers almost all the above types of events, and is formed of the attributes as shown in Table II.

TABLE II: Attributes of Events

ATTRIBUTE	DESCRIPTION
EventType	event type
EventTime	time when the event was sent
PackageName	package name of the source
MovementGranularity	movement granularity that was traversed
Action	action performed on an AccessibilityNode
ClassName	class name of the source
Text	text of the event
ContentDescription	description of the source
ItemCount	number of items that can be visited
CurrentItemIndex	index of the source can be visited
IsEnabled	if the source is enabled
IsPassword	if the source is a password field
IsChecked	if the source is checked
IsFullScreen	if the source is taking the entire screen
Scrollable	if the source is scrollable
BeforeText	the text before a change
FromIndex	the beginning of a text selection
ToIndex	index of text selection
ScrollX	the scroll offset of the source left edge
ScrollY	the scroll offset of the source top edge
MaxScrollX	the max scroll offset of the source left edge
MaxScrollY	the max scroll offset of the source top edge
AddedCount	the number of added characters
RemovedCount	the number of removed characters
ParcelableData	the Parcelable data
recordCount	the number of records contained in the event
OutBounds	bounding rectangle

Besides the problem of large volume, there are following issues that concerns the direct use of the event sequence file by our careful analysis of the event sequence.

- *Meaningless snippets.* By meaningless snippets we mean the meaningless *Activities*. An *activity* is an application component that provides a screen with which the workers can interact with. Each activity consists of a group of events. An event is generated when user interacts with the android system, such as dialling the phone, taking a

[3]http://developer.android.com/reference/android/view/accessibility/AccessibilityEvent.html

snapshot, viewing a map. But not all events are related to main business logic of the applications.

- *Redundant snippets.* Workers may wander around some specific interface of an application, which generates a large amount of duplicate activities.
- *Meaningless attributes throughout the working profile.* For a specific application, there might be some attributes that have *Default* value throughout the working profile, which means this application doesn't interfere with certain attributes during the functioning procedure.
- *Meaningless attributes for specified event type.* For certain types of event, there are some attributes that just do no count at all, this is easy to understand, for example, for the event *TYPE_WINDOWS_CHANGED*, the attribute field *IsPassword* just doesn't mean anything.

Algorithm 1: Remove Redundant Snippets

Input: Event Sequence e_p^u of a Working Profile W_p^u;
Output: Event Sequence $e_p'^u$;
Transform the e_p^u into a_p^u;
$a_p'^u$ = "";
preA = $a_p^u[0]$;
for i *from* 1 *to* N **do**
  if $a_p^u[i]$ *!= preA* **then**
    $a_p'^u$ += $a_p^u[i]$;
  end
  preA = $a_p^u[i]$;
end
Transform the $a_p'^u$ into $e_p'^u$.

Algorithm 2: Remove Meaningless Attributes

Input: Event Sequence e_p^u of a Working Profile W_p^u;
Output: Event Sequence $e_p'^u$;
Represent e_p^u as Matrix $M(K, L)$;
Reverse Matrix $M(K, L)$ into Matrix $M'(L, K)$;
for i *from* 1 *to* L **do**
  if *row i contains the same value* **then**
    remove row i;
  end
end
Obtain the new Matrix Matrix $M'(L', K)$;
Reverse Matrix $M'(L', K)$ into Matrix $M'(K, L')$;
Generate $e_p'^u$ from Matrix $M'(K, L')$.

Following above four issues, we perform the event sequence filtering. In detail, 1) for *Meaningless snippets*, we dig out some common activities from a wide range of different working profiles using the sequential patterns mining tool [10], thus to get the common activities among the android applications, which are not application-specific. 2) for *Redundant snippets*, the goal of this step is to identify and remove the continual duplicate activities, which is application-dependent. In our implementation, for continual duplicate activities we only retain a single one, the method is presented in Algorithm 1. 3) for *Meaningless attributes throughout the working profile*, the processing method is shown in Algorithm 2. 4) for *Meaningless attributes for specified event type*, we discussed with the android developers from industry to manually recognize the attributes that are of no use to certain types of events, and finally obtain the simplified format for each type of events.

B. Decision Model

The result of this component in the framework corresponds to the first part of the final summary, which tells whether bugs are found or not by the worker. Intuitively, by identifying a set of various activities and interactions with systems of workers, we can tell if the worker finds bugs or not in most cases. And follow this intuition, we pick out some of these activities as features that can be used to train a classifier for further usage. In the initial implementation of this component of *S-CAT*, we only use the following four features to train the classification model, which generates an acceptable result by our study. In fact, it is an open question of how to choose suitable features as to effectively train a classifier, as well as choosing which classification algorithm. That is, our framework is extensible in this sense. In this study, we adopt the LIBSVM [5] tool for the implementation.

The number of screenshots. Intuitively, in the design of mechanism of our crowdsourced android testing, we encourage the workers submit screenshots when they think they find a bug, thus to provide sufficient evidence for developers in debugging. In this sense, we conclude the numbers of screenshots as a strong feature of the finding of bugs.

Pause time. Based on our observation, when performing the crowdsourced testing, the workers would stop for a while when they think they find a bug, and this will result in the abnormal time slot between the events, in fact, this phenomenon follows the cognitive psychology, as a decision will always take some mental computing. Following this, we adopt the pause time as another important feature in deciding the finding of bugs. By the way, here we use a threshold in defining the *Pause*.

Time compared to other workers. This feature is intended to distinguish the consuming time of workers in conducting a crowdsourcing task, the hypothesis behind it is quite simple, when a worker spends time more than the average time of other workers on a certain crowdsourced android testing, there is a high possibility that this worker has found some bugs.

Time compared to other applications. This feature functions as a compensation to the previous one, the hypothesis is quite similar. The difference is that we use average time of a specific worker on all the testing tasks as a comparison.

As for our case, it is a typical *C*-Support Vector Classification problem [5].

C. Potential Bugs-Related Modules

By modules we mean the Android components. The goal of this part of the framework is to extract the high risk activities that would direct the developers to the source code packages that might be related to the bugs. Basically, we use a label-based method to detect the high risk modules that are potentially bugs-related. In our initial trial, three sorts of labels are adopted, i.e., the `Debug`, `Warning` and `Error` in the logcat file. `Debug` indicates the event of taking a

screenshot, with keyword *"Screenshot"* in the logcat message, and the `Warning` and `Error` with keywords *"Error"* and *"Exception"*. Along with the logcat file, there is the event sequence file, using these keywords as indicators we extract the attached timestamps, and then we use them to locate the snippets (*activities*) in the event sequence. In that way, we get the potential bugs-related modules. This method is quite intuitive and simple, and the result demonstrates its effectiveness.

Another method we are using is based on the sequential pattern mining [10]. The basis of this method is that with the evolving of the whole process of crowdsourced testing of a certain android application, more and more testing data are gathered from crowdsourced workers. To note, the gathered data that belongs to different workers are labeled with the learned result which indicate whether bugs are found or not. With this information, we separately dig out the sequential patterns from the event sequences, for both event sequences that are bugs-related and those not. Find sequential patterns (set A) in those with bugs, and find sequential patterns (set B) in those without bugs, then use $A - B$, iteratively update the library, and then use a search based method to detect the potential bugs-related snippets (activities), in order to handle the potential modules problem and the snippets problem.

Two approaches are proposed in this stage, the first is the label based method, which uses the "error", "exception" and activity of "screenshot" to identify certain modules, the other is based on the sequential mining, i.e., we mine from all the event sequence of bugs-related sequences and bug-free sequences, then for the new coming event sequence, a search based method is implemented to find the bugs-related sequences, and thus to locate the high risk modules.

D. TPD System

Our *TPD* system is a specialised Natural Language Generation (NLG) system that follows the typical architecture described by Reiter and Dale [30], it aims at describing the extracted testing process snippets of workers. Figure 5 illustrates this architecture. Conceptually, this architecture is quite intuitive: a "communicative goal" is translated from a series of facts into readable natural language sentences, known as "surface text". Three major components constitute the NLG, each of which consists of several individual steps.

The first main component is the *Document Planner*. The input to this component is a list of features that need to be communicated to a human reader. By "content determination", the document planner interprets the features and creates "messages". After the messages are generated, "document structuring" takes place which sorts the messages into a sequence that makes sense to a human reader.

The following component is the *Microplanner*, it decides which words will be used to describe each message. In "lexicalization", the microplanner assigns specific words as parts of speech in a "phrase" about each message. Typically, the subject, verb, and object for a given message are identified. Furthermore, any modifiers such as adjectives and adverbs.

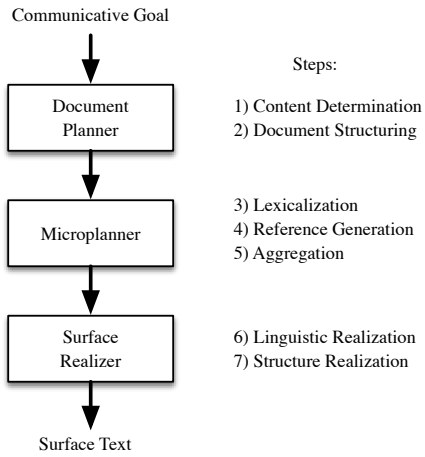

Fig. 5: The typical design of a NLG system

Next, two steps aims at smoothing the phrases so that they are more naturally read, i.e., "Reference generation" and "aggregation".

The last component is the *Surface Realizer*. The surface realizer generates natural language sentences from the phrases. Different grammar rules for the natural language dictate how the sentences are formed.Then it follows these rules to create sentences that contain the parts of speech and words given by the microplanner. These sentences are the surface text.

Our *TPD* system processes data generated from Section III-C as input, following each of the NLG steps shown in Figure 5.

Content Determination. In this step, we lock up the activities to be interpreted and extract the context of these activities. For example, what is the prerequisite activities before certain activity, thus to give hints to developers on reproducing the bugs.

Document Structuring. After generating the initial messages in the content determination phase, we put all the messages into a single document plan. We mainly follow a template based document plan, in which messages occur in a predefined order: *EventContext, ActionOfEvent*.We decide to use this order as it is natural to read.

Lexicalization. Each type of message needs a specified type of phrase to describe, i.e., in our case, for each type of event we need to decide on the words to be used to in each of the phrases. We define a set of phrase templates for each event, some of which are exemplified in Table III.

TABLE III: Event Template Examples

the user touched the *"Share"* button
the user clicked on the *"Login"* button
the user switched to the *"Content"* interface
the user scrolled down current page

Reference Generation and Aggregation. More complex and readable phrases are generated based on phrases generated from *Lexicalization* step. The working mechanism of this step

is that it looks for patterns of message types, and then groups the phrases into a sentence. For example, if two messages refer to the same component, then these two phrases of these two messages are conjoined with and "and" and the subject and verb for the second phrase is hidden. To note, the *Reference Generation* occurs alongside *Aggregation*.

Surface Realization. Finally an external library *simplenlg* [11] is adopted to realize complete sentences from the phrases formed during *Aggregation*. In the above steps, we set all words and parts of speech and provide the structure of the sentences. The *simplenlg* follows English grammar rules to concatenate verbs, and make sure the word order, plurals, and articles are correct.

E. Assemble the Summary

This part generates the final output of the summary by assembling all the above outcomes, as depicted in the example in Section **??**. There are two cases of the summary output, on one hand, the *Decision Model* decides that there are bugs found by the worker. On the other hand, the *Decision Model* decides no bugs are found by this worker.

For the first case, if there are bugs found by the worker, the final output summary will be constituted of three parts. The first part is a quick summary that tells the developers there are bugs found for this certain working profile. Then the second part presents the high risk (potential bugs-related) modules or components that are generated from Section III-C. Together with the *TPD* system introduced in Section III-D, the key testing processes of the worker are presented as the third part of the final summary.

While for the second case, it can be very simple, the summary just tells that there are no bugs found by this working profile.

IV. Evaluation

On the whole, the evaluation of *S-CAT* is binary. That is, both quantitatively and qualitatively. Mainly, we perform the quantitative evaluation on the part of the *Decision Model*, and the qualitative evaluation on the final summary.

A. Quantitative Evaluation

This part of evaluation tests to what extent our *Decision Model* can decide whether bugs are found or not, based on the working profile. In the ground truth, each working profile is labeled with 1 if bugs are found, or 0 when not.

For the evaluation, we use the metric *Accuracy*, the computation of *Accuracy* proceeds as follows. We define *hit* for a single test case as either the value 1 if the class in the test case equals the ground truth, or else the value 0. The overall *Accuracy* are defined by averaging all test cases:

$$Accuracy = \frac{\#hit}{|S_{test}|} \tag{1}$$

where $\#hit$ denotes the number of hits in the test set, and $|S_{test}|$ is the number of all test cases. As we are attempting to evaluate the whole system other than single cases, this metric well suit our scenario.

B. Qualitative Evaluation

Two major design goals behind *S-CAT* are: 1) to ease the workload of crowdsourced workers in android testing; 2) facilitate the developers in debugging. In order to evaluate how effective *S-CAT* is at achieving these goals, we endeavour to evaluate two major aspects of our framework: 1) *the experience of workers towards the summarization* and 2) *how do the developers judge the summarization*. As a result, we investigate the following research questions (RQs) as in Table IV:

TABLE IV: Research Questions

RQ_1	What information fields are considered as useful that in bug reporting of android testing?
RQ_2	Do crowdsourced workers satisfy with the summarization, in volume and in content?
RQ_3	Compared to manually written bug reports, can S-CAT provide sufficient information?
RQ_4	Are summarizations from S-CAT easier to use for deveopers in helping reproduce bugs?
RQ_5	Do developers think the S-CAT provide more hints in debugging than manually written bug reports?

The five RQs were investigated with case studies. And three sets of questions are designed to answer above research questions. In the following, we will describe the details of the case study.

Worker Experience. To assess the worker experience, we ask six different questions in the first study as to answer RQ_1 and RQ_2, as listed in Table V:

TABLE V: Questions To Assess Worker Experience

Q_1	I thought I would like to use S-CAT frequently.
Q_2	I found S-CAT unnecessary complex.
Q_3	I thought the output of S-CAT was easy to read.
Q_4	I found S-CAT very cumbersome to use.
Q_5	Which part of information do you like most from S-CAT?
Q_6	Which part of information do you like least from S-CAT?

The first four questions are answerable as "Strongly Agree", "Agree", "Disagree" or "Strongly Disagree." The last two can be answered as the "Decision Part", "Potential Modules Part", "TPD Part" or "None". The rationale behind these questions is that the output of *S-CAT* should well express the workers' testing results.

Developer Experience. To assess the developer experience, we ask five different questions in the second study as to answer RQ_4 and RQ_5, as listed in Table VI:

TABLE VI: Questions To Assess Developer Experience

Q_7	The output of S-CAT contains information that helps me understand what the bug is.
Q_8	The output of S-CAT contains information that helps me understand where the bug is.
Q_9	The output of S-CAT contains information that helps me understand how to reproduce the bug.

Also, the first three questions are answerable as "Strongly Agree", "Agree", "Disagree" or "Strongly Disagree." The ra-

tionale behind them is that the summary shall provide enough information to the developers as to help them in debugging.

Quality of Summary. And to evaluate the quality of the summary, we ask following six questions in the last study as to answer RQ_3, as listed in Table VII.

TABLE VII: Questions To Assess Quality of Summary

Q_{10}	The output of S-CAT covers all the information compared to the manually-written bug report.
Q_{11}	The output of S-CAT misses important information compared to the manually-written bug report.
Q_{12}	The output of S-CAT includes information that are unnecessary compared to the manually-written bug report.
Q_{13}	Independent of other factors, i feel that the output of S-CAT is accurate.

Each of these four multiple choice questions could be answered as "Strongly Agree", "Agree", "Disagree" or "Strongly Disagree." And the rationale behind these four questions is that the summary shall be accurate in deciding which information to present while not being too cumbersome.

We assigned values to these answers as 4 for "Strongly Agree", 3 for "Agree", 2 for "Disagree" and 1 for "Strongly Disagree." For questions $1, 3, 7, 8, 9, 10$ and 13, higher values indicate stronger performance. While for questions $2, 4, 11$ and 12, lower values are preferred. The responses to each questions are then aggregated.

V. Experiment

This section reports the results of our evaluation. First, we describe the data we are using for the experiments, and then we present the results, along with our interpretation.

A. Data

We conducted our experiments with data from our crowdsourced android testing platform, i.e., Kikbug[4]. Specifically, testing data of three applications are used to evaluate the performance of *S-CAT*, which are `SE-1800`, `UBook` and `iShopping`. `SE-1800` is an electrical monitoring application, and `UBook` is an online education application and `iShopping` is an online shopping guide application. All the three crowdsourced testing tasks were performed between the time period $October, 2015$ and $January, 2016$. In the implementation, all the information mentioned in Section II were collected through the android client and reported by the crowdsourced workers, i.e., the screenshots, the logcat file and event sequence, as well as a manually-written report. After the finish of the tasks, all the reports were validated by the developers of the applications, each was labelled as bugs found or not. Statistical information about the data we are using are illustrated in Table VIII, in which we present the number of workers taking that specific testing task, the valid report means the the manually-written report that is confirmed by the developer, and the validated bug means the worker reports a bug which is validated by the developer, the valid working profile mainly concerns about whether the logcat file and event sequence are successfully collected or not.

[4]http://kikbug.net

TABLE VIII: Statistics of The Data

App	`SE-1800`	`UBook`	`iShopping`
#Workers	132	142	173
#Valid Report	109	125	147
#Validated Bug	55	63	78
#Valid Working Profile	101	105	122

B. Participants

We had 19 participants in our case study. Fifteen were graduate students with background of Computer Science and Engineering, the remaining four were professional programmers from two different organizations, not listed due to our privacy policy. Each participant was required to review 9 summaries, we randomly chose three *S-CAT* outputs from each of the three applications. And based on the summaries, the 13 questions were answered in order to answer the aforementioned research questions. Of the 19 participants, 2 did not complete enough of the study, whom were graduate students, and their results are thrown out.

C. Results and Discussion

1) Quantitative Results: We first report the results of the accuracy of the *Decision Model* in deciding whether bugs are found or not. In the experiment, we adopted the 10-fold cross validation to train and test the classifier, as the amount of available data in our experiment is limited. And the mean average accuracy is presented in Table IX. On average, the *Decision Model* can achieve an accuracy around 0.9 on all the three applications.

TABLE IX: Average Accuracy

App	`SE-1800`	`UBook`	`iShopping`
Accuracy	0.8935	0.9167	0.9013

Currently, the realization of the *Decision Model* simply picks out some naive features that are related to the finding of bug, and adopts the SVM to train a classifier. We don't claim the improvement of the accuracy of this component as our contribution, and our framework is extensible as new classification models and new features can be imported to fulfil the task as deciding whether bugs are found or not in this step of the *S-CAT*.

2) Qualitative Results: Figure 6, Figure 7 and Figure 8 summarize the results of our case study. The box-plots in Figure 6 and Figure 7 illustrates the aggregated responses of users to questions that can be answered as as "Strongly Agree", "Agree", "Disagree" or "Strongly Disagree." And Figure 8 present the aggregated answers to which part of the *S-CAT* they like most/least, in which "DM" means the *Decision Model*, "HRM" means the component of high risk modules or say potential bugs-related modules, and "TPD" means the *TPD* module. In particular, the figures depict the answers to questions when higher values indicate stronger performance

(Figure 6), the answers to those questions when lower values are preferred (Figure 7), as well as the answers to questions when certain components of the framework shall be indicated (Figure 8).

In regard to the worker experience, we explore the results to questions $1, 2, 3, 4, 5$ and 6. From the user responses, we can tell that: 1) the users have a high tendency to use *S-CAT* (Q_1), 2) mostly, the users thought the *S-CAT* is intuitive and very easy to understand (Q_2, Q_4), and the output is easy to read (Q_3), 3) though some may not like parts of the *S-CAT* (Q_6 in Figure 8), most of the users (16 of 17) expressed their favours on certain parts of the framework (Q_6), and among the three components, the users liked the "TPD" the most (8). Based on these results we can answer RQ_1 and RQ_2 as follows: *The users are satisfied with the output of* S-CAT*, in both readability and usefulness.*

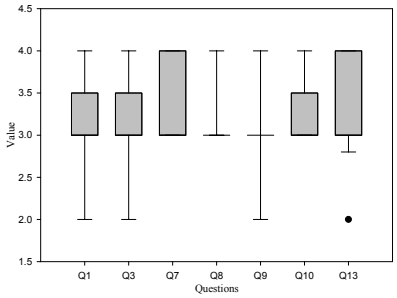

Fig. 6: Q$1, 3, 7, 8, 9, 10$ and 13

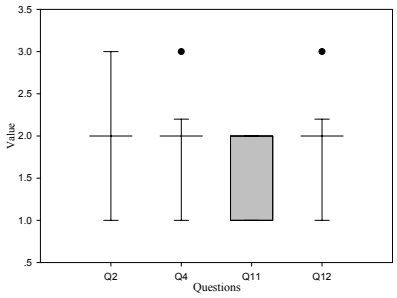

Fig. 7: Q$2, 4, 11$ and 12

Regarding the developer experience, we exploit the answers to questions $7, 8$ and 9. From the user responses, we can tell that: 1) most users agree in that the generated summary helps in understanding what the bug is (Q_7), 2) and they also hold the opinion the summary from *S-CAT* provides information in deciding where the bug is (Q_8), 3) as well as hints on how to reproduce a certain bug from the component of *TPD* (Q_9). Therefore, we can answer RQ_4 and RQ_5 as follows: *The users have a strong belief that the* S-CAT *can well facilitate the developers in understanding the bugs and locating the bugs, as well as reproducing the bugs.*

In terms of the quality of summary, we refer to the answers to questions $10, 11, 12$ and 13. From the responses of users,

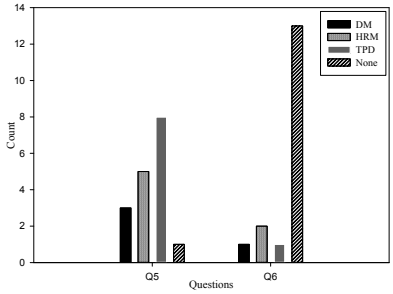

Fig. 8: Answers to question 5 and 6

we can easily get the following conclusions: 1) mostly, the information *S-CAT* provided can cover all the information compared to the manually-written report by the crowdsourced worker (Q_{10}), 2) and it covers all the important information, while not to be cumbersome (Q_{11},Q_{12}), 3) the users also agreed on that the information *S-CAT* conveyed is accurate for both the deciding of potential bugs-related modules and the description for key testing processes. Thus, we can answer RQ_3 as follows: *The users are satisfied with the quality of output from* S-CAT*, in terms of coverage and accuracy.*

VI. Related Work

There are three lines of work we would like to address as the related work, i.e., *Crowdsourced Testing*, *Summarization* and *Android Testing*.

Crowdsourcing Testing. Crowdsourcing arose with citizen science as a form of volunteerism, but now spans the gamut from work (paid microtasking) to play (games for good) [25]. Typically, crowdsourcing outsources a job to an undefined, generally large group of people in the form of an open call, which is conventionally performed by a designated agent (usually an employee) [**?**], [9], [23]. Along with this trend, crowdsourced testing is gaining more and more attention from industry. Liu et al. [20] applied crowdsourced testing in the usability testing. They studied both methodological differences for crowdsourcing usability testing and empirical contrasts to results from more traditional, face-to-face usability testing. Dolstra et al. [6] used crowdsourced testing to perform the expensive tasks on GUI testing. They use virtual machines to run the system under test and enable semi-automated GUI testing by crowdsourced workers. Nebeling et al. [26] evaluated the usability of web sites and web-based services with crowdsourcing data, where they showed that crowdsourcing data could provide an efficient and effective testing method to the web interfaces.

Summarization. Summarization has been successfully applied to many domains like news articles [28], social media [19], medical documents [1], videos [14], audio [36] and trajectory [32], in addition to technical documents. The related works closest to our approach is summarizing bug reports, which have been previous studied in [29], in which they used supervised methods to automatically generate one kind of

software artifact, bug reports, to provide developers with the benefits others experience daily in other domains. Then, Mani et al. [22] studied unsupervised summarization techniques for bug summarization, the efficacy of their unsupervised techniques were improved by applying noise identifier and filtering methods. There are also a number of other approaches studying summarizing different software artifacts and behaviors. Sridhara et al. developed novel heuristics to automatically summarize a Java method in the form of natural language comments [31]. Moreno et al. [13] proposed a summarization technique for Java classes that match one of 13 "stereotypes". Specifically, their technique selects statements from the class based on this stereotype, and then uses the approach by Sridhara [31] to summarize those statements. A most recent work by McBurney and McMillan [24] summarized the context of the source code, such as how the code is called or the output is used.

Android Testing. Besides the traditional failures due to application logic bugs, android applications often show failures that are specific of their development platform, mobile context, connectivity and so on so forth. Some specific android bugs are reported in the classification proposed by Hu et al. [15], as well as an android-specific testing technique, which is event-based and focuses on *Activity*, *Event* and dynamic type errors. Android testing has also been approached by model-based testing techniques. Takala et al. [33] described a model-based approach for android GUI testing, their technique described the GUI of an android application by state machines. An alternative approach for automatically testing an android application by its GUI was proposed by Amalfitano et al. [3], whose approach was based on a tool that explores the application GUI by simulating real user events on the user interface and reconstructs a GUI tree model. Our approach differs in that the *S-CAT* deals with crowdsourced testing, where the mobile contexts, device hardwares and system version releases are diversified.

VII. Conclusions

Crowdsourcing is an ideal solution to the android testing due to all the portability and compatibility issues. However, it meets the problem of quality control. There are two major sets of approaches that aim at the quality control in it, i.e., the design-time approaches, e.g., effective task preparation and worker selection, and the other is runtime approaches, e.g., expert review, ground truth, majority consensus, contributor evaluation, real-time support and workflow management. In this paper, we have proposed a framework *S-CAT* to summarize the crowdsourced android testing, as an answer to the problem of quality control in the specified crowdsourcing circumstance, i.e., android testing, by adopting the idea of real-time support. Our pilot evaluation results demonstrated the proposed *S-CAT* is effective in both easing the workload of workers and facilitating the developers in debugging.

References

[1] S. Afantenos, V. Karkaletsis, and P. Stamatopoulos. Summarization from medical documents: a survey. *Artificial intelligence in medicine*, 33(2):157–177, 2005.

[2] M. Allahbakhsh, B. Benatallah, A. Ignjatovic, H. R. Motahari-Nezhad, E. Bertino, and S. Dustdar. Quality control in crowdsourcing systems: Issues and directions. *IEEE Internet Computing*, (2):76–81, 2013.

[3] D. Amalfitano, A. R. Fasolino, and P. Tramontana. A gui crawling-based technique for android mobile application testing. In *ICSTW*, pages 252–261, 2011.

[4] T. W. Bickmore and B. N. Schilit. Digestor: device-independent access to the world wide web. *Computer Networks and ISDN Systems*, 29(8):1075–1082, 1997.

[5] C.-C. Chang and C.-J. Lin. Libsvm: a library for support vector machines. *TIST*, 2(3):27, 2011.

[6] E. Dolstra, R. Vliegendhart, and J. Pouwelse. Crowdsourcing GUI tests. In *ICST*, pages 332–341, 2013.

[7] S. Dow, A. Kulkarni, S. Klemmer, and B. Hartmann. Shepherding the crowd yields better work. In *CSCW*, pages 1013–1022, 2012.

[8] J. Ericsson. Ericsson mobility report, 2014.

[9] E. Estellés-Arolas and F. González-Ladrón-de Guevara. Towards an integrated crowdsourcing definition. *Information Science*, 38(2):189–200, 2012.

[10] P. Fournier-Viger, A. Gomariz, M. Campos, and R. Thomas. Fast vertical mining of sequential patterns using co-occurrence information. In *Advances in Knowledge Discovery and Data Mining*, pages 40–52. 2014.

[11] A. Gatt and E. Reiter. Simplenlg: A realisation engine for practical applications. In *ENLG*, pages 90–93, 2009.

[12] S. Greengard. Following the crowd. *Communications of the ACM*, 54(2):20–22, 2011.

[13] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *WCRE*, pages 35–44, 2010.

[14] B. Han, J. Hamm, and J. Sim. Personalized video summarization with human in the loop. In *WACV*, pages 51–57, 2011.

[15] C. Hu and I. Neamtiu. Automating gui testing for android applications. In *AST*, pages 77–83, 2011.

[16] A. Kittur, E. H. Chi, and B. Suh. Crowdsourcing user studies with mechanical turk. In *CHI*, pages 453–456, 2008.

[17] A. Kittur, B. Smus, S. Khamkar, and R. E. Kraut. Crowdforge: Crowdsourcing complex work. In *UIST*, pages 43–52, 2011.

[18] A. Kulkarni, M. Can, and B. Hartmann. Collaboratively crowdsourcing workflows with turkomatic. In *CSCW*, pages 1003–1012, 2012.

[19] Y.-R. Lin, H. Sundaram, and A. Kelliher. Summarization of large scale social network activity. In *ICASSP*, pages 3481–3484, 2009.

[20] D. Liu, R. G. Bias, M. Lease, and R. Kuipers. Crowdsourcing for usability testing. *JASIST*, 49(1):1–10, 2012.

[21] R. Longoria. Designing mobile applications: challenges, methodologies, and lessons learned. *Usability evaluation and interface design: Cognitive engineering, intelligent agents and virtual reality*, pages 91–95, 2001.

[22] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey. Ausum: approach for unsupervised bug report summarization. In *FSE*, page 11, 2012.

[23] K. Mao, Y. Yang, M. Li, and M. Harman. Pricing crowdsourcing-based software development tasks. In *ICSE*, pages 1205–1208, 2013.

[24] P. McBurney and C. McMillan. Automatic source code summarization of context for java methods. 2015.

[25] P. Michelucci and J. L. Dickinson. The power of crowds. *Science*, 351(6268):32–33, 2016.

[26] M. Nebeling, M. Speicher, M. Grossniklaus, and M. C. Norrie. *Crowdsourced web site evaluation with crowdstudy*. 2012.

[27] A. J. Quinn and B. B. Bederson. Human computation: a survey and taxonomy of a growing field. In *CHI*, pages 1403–1412, 2011.

[28] D. R. Radev, S. Blair-Goldensohn, Z. Zhang, and R. S. Raghavan. Newsinessence: A system for domain-independent, real-time news clustering and multi-document summarization. In *HLT*, pages 1–4, 2001.

[29] S. Rastkar, G. C. Murphy, and G. Murray. Summarizing software artifacts: a case study of bug reports. In *ICSE*, pages 505–514, 2010.

[30] E. Reiter, R. Dale, and Z. Feng. *Building natural language generation systems*, volume 33. MIT Press, 2000.

[31] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *ASE*, pages 43–52, 2010.

[32] H. Su, K. Zheng, K. Zeng, J. Huang, S. Sadiq, N. J. Yuan, and X. Zhou. Making sense of trajectory data: A partition-and-summarization approach. In *ICDE*, pages 963–974, 2015.

[33] T. Takala, M. Katara, and J. Harty. Experiences of system-level model-based gui testing of an android application. In *ICST*, pages 377–386, 2011.

[34] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *NIST*, 7007(011), 2002.

[35] B. A. Vander Schee. Crowdsourcing: Why the power of the crowd is driving the future of business. *Journal of Consumer Marketing*, 26(4):305–306, 2009.

[36] A. Waibel, M. Bett, F. Metze, K. Ries, T. Schaaf, T. Schultz, H. Soltau, H. Yu, and K. Zechner. Advances in automatic meeting record creation and access. In *ICASSP*, volume 1, pages 597–600, 2001.

[37] D. Zhang and B. Adipat. Challenges, methodologies, and issues in the usability testing of mobile applications. *IJCHI*, 18(3):293–308, 2005.