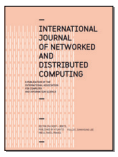
International Journal of Networked and Distributed Computing
Vol. 7(4); September (2019), pp. 167–174
DOI: https://doi.org/10.2991/ijndc.k.190917.002; ISSN 2211-7938; eISSN 2211-7946
https://www.atlantis-press.com/journals/ijndc



Research Article

SAIFU: Supporting Program Understanding by Automatic Indexing of Functionalities in Source Code

Masashi Nishimoto*, Keiji Nishiyama, Hideyuki Kawabata, Tetsuo Hironaka

Graduate School of Information Sciences, Hiroshima City University, Hiroshima, Japan

ARTICLE INFO

Article History

Received 14 March 2019
Accepted 12 May 2019

Keywords

Program understanding
dataflow graph
tag cloud
event-driven programming

ABSTRACT

Programs in the event-driven style that are typical of mobile and/or Web applications are becoming complex and hard to maintain. For the purpose of reducing the burden put on software developers while reading source code to understand its details, we propose a tool for supporting program understanding, named *SAIFU* (a tool for Supporting program understanding by Automatic Indexing of Functionalities). SAIFU automatically extracts implemented functionalities from source code and puts annotations to them. SAIFU helps the user grasp the behavior and the structure of a whole program by showing a list of the annotations of functionalities. SAIFU highlights a set of statements of the source code that are related to any functionality on the annotation list so that the user can investigate the implementation details of a particular functionality. Experimental results obtained by applying SAIFU to 16 applications in Google Samples confirm that the tool is effective for finding out important statements from existing Android application programs.

1. INTRODUCTION

Application software is a complex mixture of functionalities. The complexity is, for one thing, due to the event-driven style of software for mobile and/or Web applications where each functionality constituting the software is implemented by combining descriptions that are scattered all over the source code, i.e., each functionality is not clearly separated in the source code. Such complexity of software structure is a serious obstacle to the smooth and safe modification and maintenance of software.

There are various ways to help you understand the source code [1–8]. For example, a high-performance editor that allows syntax highlighting or pretty printing, and an integrated development environment with a functionality to offer an easy way to refer the definition part of each identifier in the source program cannot only be used as a tool for software development but also be usable as a tool to support program understanding. In addition, tools such as debuggers[1] and profilers,[2] which are indispensable tools for analyzing program behavior and solving problems, can also be regarded as tools for supporting program understanding in a broad sense. However, as far as we know, there is no tool that can directly support the user understand what is (and how) written in source programs.

For the purpose of reducing the burden put on software developers while reading source code to understand its structure and the details, we propose a tool for supporting program understanding, named *SAIFU*. SAIFU automatically extracts implemented functionalities from source code and puts annotations, which we call *summaries*, to them. SAIFU lets the user focus on the statements of source code and check the implementation details corresponding to each functionality. SAIFU helps the user grasp the behavior and the structure of a whole program by showing a clickable list of the annotations of functionalities.

Extracting individual functionalities is carried out by separating and relating program elements in the source code by constructing dependency graphs. Each program element corresponds to a node of an abstract syntax tree that denotes the source code. Since closely related parts in a program are connected by dependency relations, such as the producer–consumer relation of data and the relation between two method calls where one is a generator of an object and the other is a method that belongs to the object, dependency graphs consisting of program elements reveals sets of program lines, each of which implements a particular functionality.

Although dependency analysis of ordinary monolithic applications might yield a huge single dependency graph where all statements in the program would be included, *event-driven applications* that are typical of mobile applications and Web applications are not the case. On the other hand, event-driven applications usually consist of method definitions that bundle several method calls that are unrelated to each other. SAIFU can extract functionalities beyond the borders of those method definitions.

An extracted set of program elements almost always include a set of API method calls, where Application Programming Interfaces (APIs) provide commonly used software components. Names of API methods usually possess useful information for grasping their behaviors. In addition, names of packages and classes to which those methods belong can be utilized to extract a suggestive summary of the part of the program from which the set of program elements

*Corresponding author. Email: nishimoto.masashi@ca.info.hiroshima-cu.ac.jp
[1]GDB: https://www.gnu.org/software/gdb/, jdb: http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html
[2]Valgrind: http://valgrind.org/, VisualVM: https://visualvm.github.io

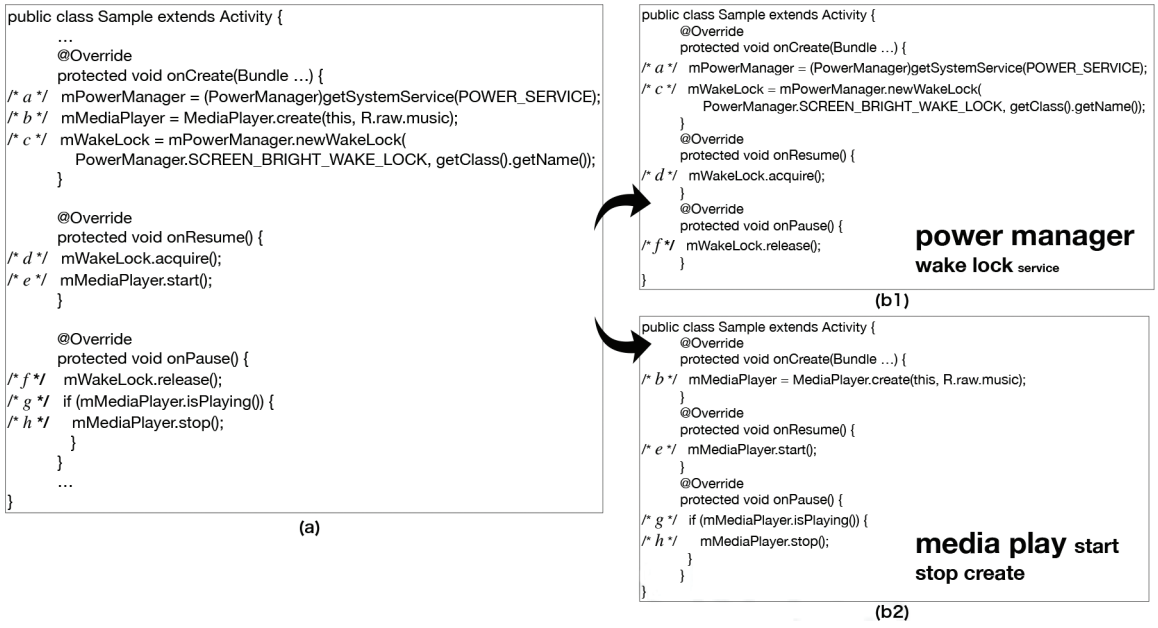


Figure 1 | Typical example of an application program for Android.

are obtained. Because a plain set of words might not be useful as an annotation for a part of a program, SAIFU weights each extracted word to generate a *tag cloud* to represent a summary of the program part so that frequently used words and apparently characteristic words for the part are emphasized.

In this paper, we describe the design of SAIFU and its prototype implementation for Android applications. We also show the results of subjective evaluation based on the prototype to confirm the effectiveness of the proposed system.

The rest of the paper is organized as follows. In Section 2, we present an overview of SAIFU by showing a motivating example. In Section 3, we describe the design and implementation of SAIFU. In Section 4, we discuss the usability of SAIFU. Section 5 shows a summary of related work.

2. OVERVIEW OF SAIFU

2.1. Motivation and Aims

Figure 1a shows a code segment of a typical Android application program. You can see three method definitions in the program. However, in a sense, each of the three is not implementing a separate functionality; those method definitions are used just for handling incoming events. The fact that the borders that divide a source program into separate functionalities do not coincide with the borders of methods, which is common for event-driven programs, makes it hard to read and maintain existing programs.

The program shown in Figure 1a is virtually a merger of those shown in Figure 1b1 and b2. When you are to read a lengthy source program to understand its behavior, you might have to separate the descriptions related to each functionality that may concern in your head, like separating Figure 1a into Figure 1b1 and b2. The burden posed to the developer is summarized as follows:

- It is not easy to pinpoint where and how each functionality is implemented in the source code.
- It is not easy to grasp what functionalities are implemented in the source code and how the program behaves.

These are what we want to eliminate by offering a tool named SAIFU, to help the user read and understand existing event-driven style programs.

In the rest of this section, we describe the GUI of SAIFU to show the usage, the essence of the functionality extraction, and the idea of annotation generation, in each subsection.

2.2. GUI of SAIFU

Figure 2 shows a screenshot of SAIFU's GUI, illustrating the scene where a source file of AccelerometerPlayActivity.java from Google Samples[3] is loaded. SAIFU works as a source code browser which has been implemented to be used through a Web interface. Each line of the opened source file is displayed on the right pane as shown in Figure 2.

When a source file is opened, SAIFU analyzes the source code and extracts relations among program elements such as method calls and variable references. Then, SAIFU extracts sets of program elements that are closely connected by such as data dependency relations and object-belonging method relations. We call each extracted set here a *functionality*; we expect that the elements in a set are used to implement a particular functionality in an ordinary sense.

Once a set of functionalities are extracted from the source code, SAIFU constructs an annotation for each functionality in the form of a tag cloud. The annotations are listed on the left pane as shown in Figure 2. By examining the annotations, the user would be able to grasp what functionalities are implemented in the source code. It would not be difficult to find out, for example, that the fourth and the fifth items of the list in Figure 2 are related to code segments for managing a WakeLock component and an acceleration sensor, respectively.

[3] https://github.com/googlesamples

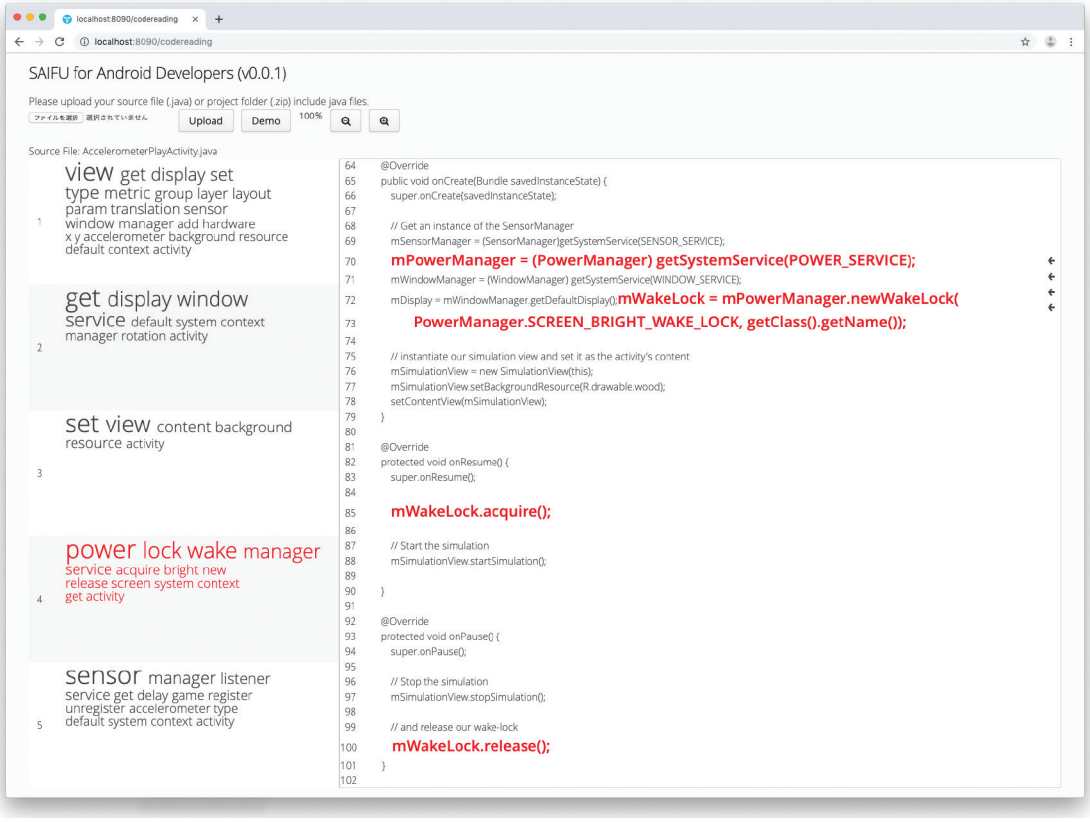


Figure 2 | Source Code Browser (SCBrowser) of SAIFU displaying AccelerometerPlayActivity.java.

The list of annotations is clickable. When a tag cloud is selected (clicked by mouse), the corresponding statements of the source code related to the annotation are highlighted on the right pane. In the case of Figure 2, the fourth tag cloud should be the right choice if the user wished to check the source code description related to the WakeLock management. In Figure 2, related lines to the fourth annotation are highlighted and are easily located. As such, SAIFU helps the user detect lines related to a particular functionality to inspect implementation details, as well as to search coding patterns.

2.3. Extracting Functionalities from Source Code

To extract groups of statements where each group is related to separate functionality, we use a dependency analysis to obtain statements that interact with common objects beyond method boundaries. While analyzing, relations between program elements such as data dependency relations and object-belonging method relations are extracted from source code. The relations can be represented as dependency graphs. A dependency graph consists of two kinds of nodes and directed edges that link nodes of different kinds. Each node of a dependency graph represents either an object (an instance of a class or a value of a basic type) or an API method. Figure 3 shows the dependency graphs obtained from the source code in Figure 1a.

A statement described in the source code is expressed by a set of nodes connected by edges. For example, the directed edge labeled with *gen* from the node Activity.getSystemService to the node mPowerManager expresses a part of the statement *a* in Figure 1a. Two statement sets $\{a, c, d, f\}$ and $\{b, e, g, h\}$ are identified from two separate graphs in Figure 3. These two statement sets are consistent with the statements appearing in programs b1 and b2 in Figure 1.

2.4. Summarizing Functionalities

Each extracted group of statements consists of API method calls. Since the name of an API method usually indicates its meaning and behavior, a set of words extracted from API method names used in the group of statements might be used for expressing the behavior of the set of statements well. We extract words from API method names including names of packages and classes to construct a readable and suggestive summary of the part of the program from which the set of statements are obtained.

In a tag cloud, the importance of each word is represented by its size and place. Basically, words with high frequency should be emphasized. However, common words like “get” and “set” should not be treated as important words. To balance the weighting, we use the Term Frequency-Inverse Document Frequency (TF-IDF) measure which has been commonly used in the field of document search. From the examples annotations shown in Figure 1, the approach for constructing tag cloud annotations seems to be appropriate.

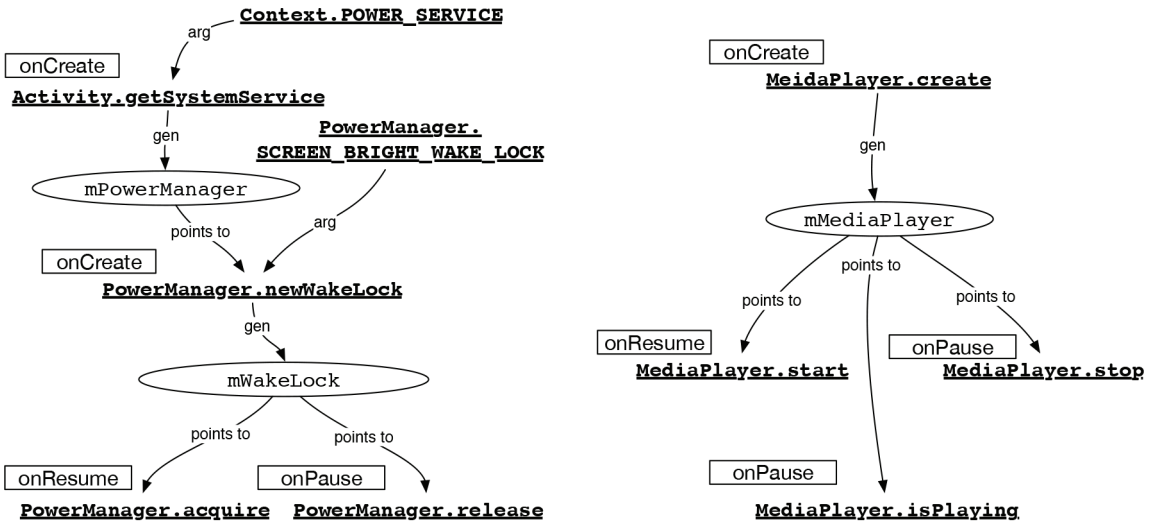


Figure 3 | Dependency graph corresponding to the source code in Figure 1.

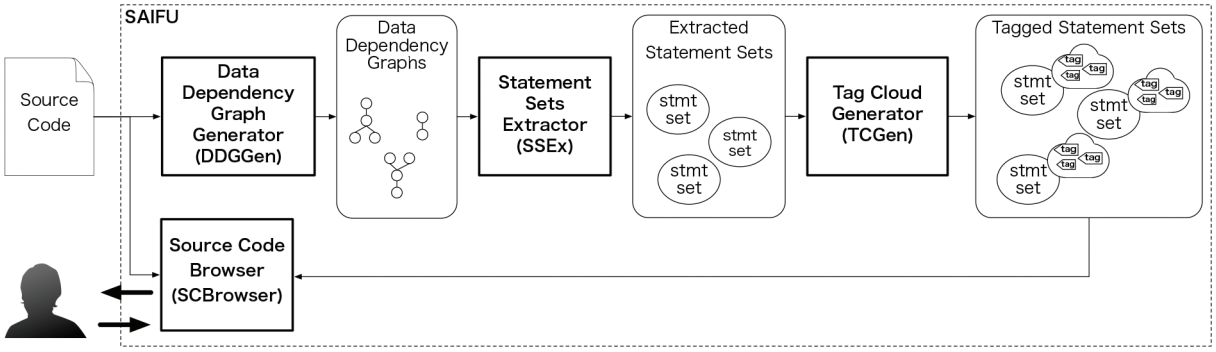


Figure 4 | Structure of the program understanding support tool, SAIFU.

3. DESIGN AND IMPLEMENTATION OF SAIFU

3.1. Structure of SAIFU

SAIFU works as a source code viewer that shows detailed information on one given source file. Figure 4 shows the structure of SAIFU. As shown in the figure, SAIFU consists of four components: the *Data Dependency Graph Generator* (DDGGen), the *Statement Sets Extractor*, the *Tag Cloud Generator* (TCGen), and the *Source Code Browser* (SCBrowser). SAIFU is controlled via SCBrowser, a GUI of the system. When a source file is given, SAIFU analyzes it to produce statement sets, makes annotations to each set and displays information of the source program in the SCBrowser's screen.

SAIFU processes one source program at a time as follows. First, data dependency graphs are generated from the source code. Then, statement sets are extracted based on the graphs. Each statement set is then annotated by a tag cloud that is generated from the statement set itself. Annotated set of statements are, together with the original source code, supplied to the SCBrowser.

In the rest of this section, the implementation details of each component of SAIFU is described.

3.2. Data Dependency Graph Generator

Data Dependency Graph Generator (DDGGen) extract subtrees of the abstract syntax tree of the source code, where each subtree corresponds to a statement.[4] Then, DDGGen transforms each extracted subtree into basic dependency graphs based on the following rules:

- Elements on the left- and the right-hand side of an assignment statement are under *gen*-relation.
- A reference to a variable that points to an object and the invocations of its belonging methods are under *points to*-relation.
- A method invocation and references to its arguments are under *arg*-relation.

Finally, basic dependency graphs are mutually connected to make a small number of larger graphs.

Figure 5 shows the abstract syntax tree of the program in Figure 1a. The tree is composed of subtrees shown in Figure 6a that correspond to statements. Eight basic dependency graphs shown in

[4]Including a few special elements such as conditional expression used for if- or for-constructs.

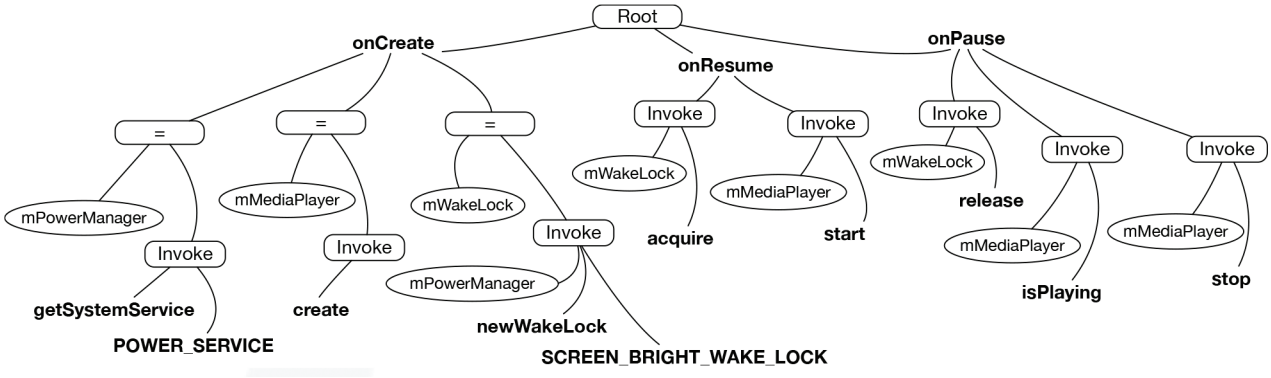


Figure 5 | Abstract syntax tree of the program in Figure 1a.

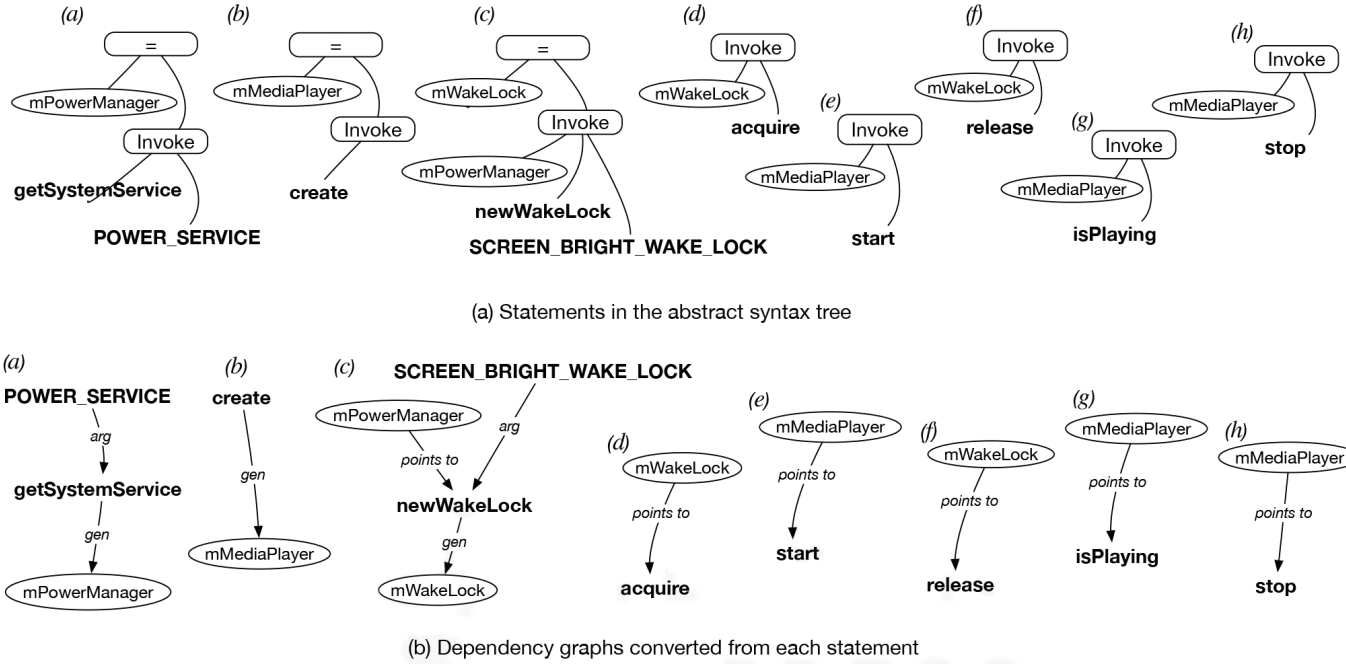


Figure 6 | Statements extracted from Figure 5 and transformed data dependency graphs.

Figure 6b are obtained from Figure 6a and connected to make the graphs shown in Figure 3.

3.3. Statement Set Extractor

Statement Set Extractor collects statements related to each connected component of the data dependency graphs generated by the DDDGen. This is done by gathering invoked methods in the graphs. Location information of their invocation sites is attached to each set.

3.4. Tag Cloud Generator

Tag Cloud Generator is two-phased; extracting words from statement sets and weighting them to construct tag clouds.

In the first phase, first, API method names are cut into smaller pieces by parsing those names taking common forms of combined words such as camel case and snake case into account. For example, the sets of words {power, manager} and {number, elements} are obtained from PowerManager and number_elements, respectively. Next, each word is normalized by using the morphological analysis to remove variations in word forms (e.g., "started" and "elements" are normalized to "start" and "element", respectively).

Each set of words are arranged in the second phase to make a tag cloud. The position and the size of each word in a tag cloud are determined by calculating the TF-IDF value among all word sets. During the process, stop words such as in and on are excluded. Words of the same TF-IDF values are ranked in the lexicographical order.

3.5 Source Code Browser

The Source Code Browser displays a list of tag clouds. The list of tag clouds is sorted in the descending order of the number of components in each tag cloud annotation.

Table 1 | Sixteen programs from Google Samples

Name of the Android project	Name of the source file	LOC	M	VP1	VP2
AccelerometerPlay	AccelerometerPlayActivity.java	430	22	×†1	✓
BasicNetworking	MainActivity.java	130	5	×†2	✓
Camera2Basic	Camera2BasicFragment.java	1030	41	✓	✓
ActionBarCompat-Basic	MainActivity.java	88	3	✓	✓
ActionBarCompat-ListPopupMenu	PopupListFragment.java	135	6	✓	✓
ActionBarCompat-ShareActionProvider	MainActivity.java	210	11	✓	✓
ActionBarCompat-Styled	MainActivity.java	81	5	✓	✓
ActiveNotifications	ActiveNotificationsActivity.java	67	5	✓	✓
ActiveNotifications	ActiveNotificationsFragment.java	194	7	×†1	✓
ActiveNotifications	MainActivity.java	111	5	×†2	✓
ActiveNotifications	SampleActivityBase.java	53	3	×†2	✓
ActiveNotifications	LogFragment.java	109	4	×†1	✓
ActivityInstrumentation	MainActivity.java	111	1	✓	✓
ActivitySceneTransitionBasic	DetailActivity.java	160	5	✓	✓
ActivitySceneTransitionBasic	MainActivity.java	130	6	✓	✓
AppRestrictionEnforcer	AppRestrictionEnforcerFragment.java	593	34	×†1	✓

LOC: Lines of code, M: Number of method definitions.

3.6. A Prototype Implementation of SAIFU

We have implemented a prototype of SAIFU. The prototype is described in Java language. To construct the DDGGen, Java parser library of Eclipse Java development tools[5] was used. Stanford Core NLP[6] was also used for the morphological analysis in TCGen. SCBrowser[7] of SAIFU was implemented as a Web application by using the SpringBoot framework.[8]

The elapsed time spent to process the source file and display the result on the SCBrowser's screen was, in the case of Figure 2, <2 s, which is acceptable.

4. EVALUATION

To evaluate the usefulness of SAIFU, we apply it to some Android programs and investigate the results. We evaluate SAIFU from the following viewpoints: the appropriateness of the functionality extraction capability from source code, the quality of the annotations in the form of tag clouds, and the usability of the system as a whole.

We apply SAIFU to 16 sample programs taken from Google Samples in January 2017. Table 1 shows an overview of them. Each number in columns labeled LOC and M represents the number of lines and the number of method definitions in the source code, respectively.

Columns labeled VP1 and VP2 are explained in the following sections.

4.1. Appropriateness of the Functionality Extraction Capability of SAIFU

SAIFU extracts *functionalities* implemented in a source file and represents them as sets of statements. To evaluate the appropriateness of the extraction method, we examine each extracted statement sets.

As described in Section 2.2 and illustrated in Figure 2, many of extracted statement sets are intuitively judged as valid. In fact, nine out of 16 source files shown in Table 1, including Camera2Basic whose source file was longer than 1000 lines, were considered to be appropriately processed in the sense that they almost agree with what was understood from comments written in the source files.

A few files listed in Table 1 are not observed as perfectly treated. For example, in the case of AccelerometerPlay, an extracted set of statement sets were too large to be called as "the set of sentences that implements one functionality." The corresponding tag cloud annotation to the large set of statements is shown at the top of the list displayed in the left pane in Figure 2. In fact, the annotation hints that the set of statements performs setting views on the screen, managing sensors, generating layouts of displayed objects, calculating coordinates of objects, and so on. Actually, the application shows "iron balls" that keep rolling on the screen whose motions are affected by the gravity, tilt and the speed and acceleration of the motion of the smartphone device; many aspects are tightly connected to realize displaying moving iron balls. The same situation was observed for those applications marked ×†1 in Table 1.

It is obvious that any "functionality" can be seen as a combination of the set of smaller subfunctionalities. Simple application of the dependency analysis to the source code tend to generate very large sets of statements. SAIFU's functionality partitioning capability apparently has room for improvements.

Another situation where the extracted sets of statements were not appropriate, observed for those applications marked ×†2 in Table 1, was caused by the limitation of the current version of SAIFU; the incapability of dealing multiple source files simultaneously.

4.2. Quality of Annotations as Tag Clouds

Extracted functionalities are annotated by tag clouds by SAIFU. As illustrated in Figure 2, extracted words and their weights put by SAIFU are considered to be appropriate on the whole and consistent with what is described in the source code.

To evaluate the property of SAIFU's tag cloud generation in detail, we compare tag cloud annotations generated by SAIFU with the output generated by using a simpler method, which weights each word depending only on its frequency. Note that SAIFU is based on

[5]https://www.eclipse.org/jdt
[6]https://stanfordnlp.github.io/CoreNLP
[7]Available at http://capis.ca.info.hiroshima-cu.ac.jp:8090/codereading
[8]https://projects.spring.io/spring-boot

the TF-IDF measure that takes not only frequency of a word but also the "peculiarity" or "scarcity" of the word among all sets of words into account while calculating the weight put to the word. Figure 7 shows the top five tag cloud annotations generated by using the two methods out of the sets of sentences extracted from BasicNetworking. We can see several differences between the lists shown in Figure 7a and b.

- In Figure 7a, common words such as "get" and "find" are treated as less important.
- In Figure 7b, tag clouds made of equally frequently appearing words are not treated appropriately, as in the fifth tag cloud.

Note that, in the tag cloud annotations shown in Figure 2, "get" and "set" appear at the first positions in some tag clouds. This is because, in the case of AccelerometerPlayActivity.java, the fact that those words appear very frequently in the corresponding sets of words while the number of sets of words that include "get" or "set" is small. It might be preferable to treat "get" and "set" as stop words to exclude in each word set.

Source programs are not ordinary text documents. Thus, ranking each code segment or set of statements by using TF-IDF for searching might not be the best way. However, ranking each word in a set of statements based on the TF-IDF measure seems to be preferable.

4.3. Usability of SAIFU

There should be many cases SAIFU would be useful. For example, when you are to

- read unfamiliar source files and grasp what functionalities are implemented in them and how,
- modify or refactor existing lengthy programs, or
- find coding patterns for implementing some functionalities by investigating a bunch of source files obtained from somewhere on the Net.

To give a concrete example, suppose we are to modify AccelerometerPlay's source code to dim the device's screen by managing a WakeLock component. In this case, it would be suitable to check the part related to the fourth tag cloud annotation in Figure 2. When we click the annotation, SAIFU would display the statements in the source code that are related to the management of a WakeLock, and the user could confirm the highlighted statements on the source code. Concretely, lines 70, 72, 73, 85, and 100 are related to the annotation. By looking through these small number of lines, it would be easy to find out that the named constant PowerManager.SCREEN_DIM_WAKE_LOCK referred to on line 73 should be modified.

5. RELATED WORK

Several tools have been studied to support the understanding of programs. Myers [9] classifies the various visualization systems for program understanding into four categories: static code visualization, static data visualization, dynamic code visualization, and dynamic data visualization. We introduce related studies based on these four categories.

Rigi [1] is a static code visualization system which extracts software elements such as functions and variables from source code and visualizes their relationships (e.g., function calls and data reference relations). SHriMP [2] extends Rigi [1] by introducing a hierarchical display mechanism and Whorf [3] presents information

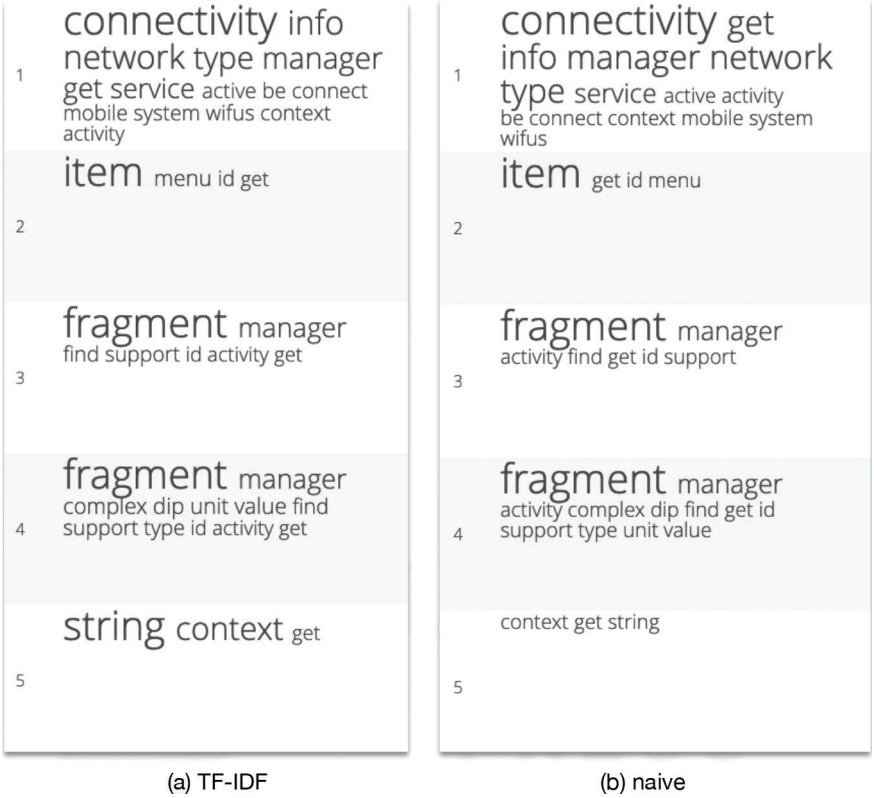


Figure 7 | Comparison of annotations in the tag cloud generated from MainActivity.java.

between software components by multiple views help understand programs of large software. SAIFU is also a static code visualization tool. SAIFU does not support graphical presentations but supports direct and summarized text-based view of source code.

Code Crawler [4] and Pinzger et al. [5] are systems that visualize static data, i.e., visualize the dependency between objects and the relation between classes and packages. SAIFU does not visualize data dependencies but uses the information to extract functionalities consisting of program elements.

BALSA [6] and PV [7] are systems for dynamic code visualization. These tools illustrate how the values of variables change while showing the current program execution location on the source code.

Jeliot 3 [8] is a dynamic data visualization tool that visualizes how variables are changed by program execution. SAIFU does not use the dynamic information but shows a kind of program slices that could be of help for grasping behaviors of programs.

Some visualization systems do not fall into these four categories. Code Canvas [10] can arrange the layout of multiple source code editors to present effective visualizations so that various source code can be smoothly moved between related source codes. CodeCity [11] and CodeForest [12] are tools that visualize software metrics such as the number of lines of the source code, the number of classes and methods to be called, and so on. These studies use a three-dimensional graphical representation that is intuitively understandable by grasping the structure and characteristics of the program. SAIFU currently does not support editing or measurement of code. Integration of those functionalities to SAIFU should be straightforward.

6. CONCLUSION

In this paper, we described the design and implementation of SAIFU that supports program understanding. SAIFU is expected to be useful for developers who are to maintain and expand existing programs. The result of a subjective evaluation confirms that SAIFU is applicable for understanding event-driven applications. Future work includes the improvement of the preciseness of functionality extraction, the refinement of weighting for better tag cloud representations, and the extension of analyzing across multiple source files.

CONFLICTS OF INTEREST

The author declares they have no conflicts of interest.

REFERENCES

[1] H.A. Muller, K. Klashinsky, Rigi: a system for programming-in-the-large, Proceedings of the 10th International Conference on Software Engineering, (ICSE) IEEE, Singapore, 1988, pp. 80–86.
[2] M.D. Storey, H.A. Muller, Manipulating and documenting software structures using SHriMP views, Proceedings of International Conference on Software Maintenance, IEEE, Opio, France, France, 1995, pp. 275–284.
[3] K. Brade, M. Guzdial, M. Steckel, E. Soloway, Whorf: a visualization tool for software maintenance, Proceedings of IEEE Workshop on Visual Languages, IEEE, Seattle, WA, USA, USA, 1992, pp. 148–154.
[4] M. Lanza, S. Ducasse, H. Gall, M. Pinzger, CodeCrawler - an information visualization tool for program comprehension, Proceedings of the 27th International Conference on Software Engineering (ICSE), IEEE, Saint Louis, MO, USA, 2005, pp. 672–673.
[5] M. Pinzger, K. Graefenhain, P. Knab, H.C. Gall, A tool for visual understanding of source code dependencies, 2008 16th IEEE International Conference on Program Comprehension, IEEE, Amsterdam, Netherlands, 2008, pp. 254–259.
[6] M.H. Brown, R. Sedgewick, A system for algorithm animation, SIGGRAPH Comput. Graph. 18 (1984), 177–186.
[7] D.A. Kramlich, G.P. Brown, R.T. Carling, P. Souza, C.F. Herot, Program visualization: graphical support for software development, Computer 18 (1985), 27–35.
[8] A. Moreno, N. Myller, E. Sutinen, M. Ben-Ari, Visualizing programs with jeliot 3, Proceedings of the Working Conference on Advanced Visual Interfaces (AVI), ACM, Gallipoli, Italy, 2004, pp. 373–376.
[9] B.A. Myers, Visual programming, programming by example, and program visualization: a taxonomy, Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI), ACM, Boston, MA, USA, 1986, pp. 59–66.
[10] R. DeLine, K. Rowan, Code canvas: zooming towards better development environments, Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE), ACM, Cape Town, South Africa, 2010, pp. 207–210.
[11] R. Wettel, M. Lanza, R. Robbes, Software systems as cities: a controlled experiment, Proceedings of the 33rd International Conference on Software Engineering (ICSE), ACM, Waikiki, Honolulu, HI, USA, 2011, pp. 551–560.
[12] K. Maruyama, T. Omori, S. Hayashi, A visualization tool recording historical data of program comprehension tasks, Proceedings of the 22nd International Conference on Program Comprehension (ICPC), ACM, Hyderabad, India, 2014, pp. 207–211.