

IIAnalyzer: A Reverse Engineering Tool to Analyze Interactions between Objects

Toshihiro Kamiya^{†‡}

[†]*Presto, Japan Science and Technology Agency (JST)*

[‡]*National Institute of Advanced Industrial Science and Technology (AIST)*

t-kamiya@aist.go.jp

Abstract

The tool named IIAnalyzer is a reverse engineering tool for programs written in Java. By identifying and analyzing interactions between the objects that are instantiated in a program execution, the tool presents the user with diagrams helpful in understanding a dynamic structure and characteristics of the target program. The current version provides the following functions; i) detection of similar behaviors in an execution, ii) localization of functions in a target program (feature analysis). In this paper, I will represent the former function of the IIAnalyzer, by describing the detection algorithm and the diagram used to express the similarity and difference of the behaviors.

1. Introduction

During an execution of object-oriented program, a number of objects are instantiated and interact with each other, and such collaboration emerges as functions of the program. The development and maintenance of the program requires the developer to understand these objects and their interactions. A number of development methods and technologies support the modeling, designing, and understanding of objects and interactions, e.g., UML (Unified Modeling Language)[10] provides collaboration diagram and sequence diagram that are used to describe objects and their relations. Some reverse engineering tools generates a sequence diagram from a program execution.

The dynamic nature of object-oriented programming languages makes it harder to understand object-oriented programs with only static analysis methods. A dynamic dispatch, which is commonly used in object-oriented programs, often makes it impossible to determine the precise behavior of a

program, without the execution context of it. A program slicing is a typical example. The backward slice aims to determine the set of statements that effects a value of a variable in an execution point (criterion), and the size of a slice tends to be larger when you use static slicing[6] technique than dynamic slicing[1]. The reason is that the dynamic slicing takes account of what method body was called in each dispatch in the given execution and excludes the method bodies that were not called, while the static slicing takes account of all of possibly-called methods at each call statement. Another example is software metric. In recent years, dynamic metrics have been proposed to capture dynamic characteristic of object-oriented program[2][4]. Also, feature analysis (or feature location), which performs mapping a feature into a sub system of software, can be performed either in a static way[3] or in a dynamic way[5].

The tool named IIAnalyzer is a reverse engineering tool for programs written in Java. By identifying and analyzing interactions between the objects that are instantiated in a program execution, the tool presents the user with diagrams helpful in understanding a dynamic structure and characteristics of the target program. The current version provides the following functions; i) detection of similar behaviors in an execution, ii) localization of functions in a target program (feature analysis). In this paper, I will represent the former function of the IIAnalyzer, by describing the detection algorithm and the diagram used to express the similarity and difference of the behaviors. The detection of similar behavior enables the users to investigate the functional similarities in source code of a program.

2. Similarity of behavior

In this study, the similarity of behavior is detected from an execution of a program. Here, an execution

trace is a list of entering a method and exiting from a method of each thread (Fig. 1). It contains class name, method name, signature, and ID of calle object of a method call.

```

1:A#static main(String[]) {
2: System.out PrintStream#println(String)void {
3: }
4: obj1 A#say(int)void {
5:   System.out PrintStream#print (String)void {
6:   }
7:   System.out PrintStream#println(String)void {
8:   }
9: }
10: obj1 A#say(int)void {
11:   System.out PrintStream#print (String)void {
12:   }
13:   obj2 B#say()void {
14:     System.out PrintStream#println(String)void {
15:     }
16:   }
17: }
18: System#static exit(int)void {
19: }
20:}

```

Figure 1. An outline of a sample execution trace

When given two (sub-) sequences in an execution are calling the identical methods of the identical objects in the identical order, the two sequences are considered identical (hereafter, “strictly equivalent”). In an execution of a program, such similar behaviors often occur by a loop, a recursive call, and a duplicated code. On the other hand, the behavior of the same code sometimes varies when the execution tracks the code twice and falls into different branches at the conditional statements in the code.

The strict equivalence seems too strict in some kind of analysis at high abstraction level, e.g., investigating functional similarities. By following the definition of strict equivalence, two sequences tracking the identical loop in different times are not equivalent, but such slighter difference can be neglected in this kind of analysis. Assuming that a sequence in a method M calling N via D and a sequence in M calling N directly. If the D is a simple delegation to N and does hardly anything except for it, the two sequences should be determined identical.

3. The algorithm to extract similar behaviors from an execution trace

In this paper, I propose a more relaxed equivalence relation named “unit-operation equivalence”, which means that given two sequences execute the identical set of unit operations either directly or indirectly. Here, a unit operation is a basic operation or an atomic one. In concrete terms, a unit operation is a method from the basic library of the programming language or a

user-defined method that does not call other methods. In case of Java, the methods of classes from JDK (Java development kit) library, such as a class String, are unit operations.

Additionally, in this paper, a target sequence of the similarity detection is a (sub-) sequence which begins at entering of a method and ends at exiting from it. This limitation is not essential to the detection algorithm, but makes it easier to illustrate the detected similarities, because both a target and a unit operation are represented by a method call. Such a diagram helps intuitive understanding of similarity and difference between behaviors.

The similarity detection algorithm is the followings:

1. For each method invoked in an execution trace, if the method is from a basic library or if the method does not call other methods, mark the method as “unit operation”.
2. For each sequence m corresponding to a method invocation, that is, a sequence which starts at a entering a method and ends at a exiting from it, find invocations of “unit operations” in the sequence and make a set of these unit operations, $UO(m)$.
3. For each pair of sequences (m, n) , if $UO(m)$ is the same as $UO(n)$, then the pair (m, n) is a unit-operation equivalence.
 - 3-1. If the sequence m is part of the sequence of n in the trace, then the equivalence relation (m, n) is labeled as “delegation”.

The reason for distinguishing a delegation from the other equivalence relations is that it tends to amount to a large part of the detected equivalence relations and does not seem so interesting for some applications.

4. The diagram to visualize similarity and difference between behaviors

This section presents a diagram named multi-layered summarized instance call graph (hereafter, MSI-CG), which is proposed to visualize the similarities of behaviors detected by the above-mentioned algorithm.

A node of MSI-CG, which is labeled by a class name and a method name, means a (sub-) sequence which begins at entering the method and ends at exiting from it. A node sometimes corresponds to two or more invocations of the method in the execution trace. A directed arc, which means a method call, is running from the caller to the callee. Each node is colored with either gray (if the method is a unit

operation) or white (otherwise). Also, shape of each node is either rectangle (if the method is calling all of the unit operations in the graph, directly or indirectly) or ellipse (otherwise). The arc is either solid (if the callee method always invokes the caller method) or dotted (if the callee method sometimes does not call the callee method). Fig. 2 shows an example of the MSI-CG. The interpretation is as follows: there are sequences calling class A's method say (hereafter A#say) and they are unit-operation equivalent. One of the sequences calls both PrintStream#print and PrintStream#println directly, while the other calls PrintStream#print directly but calls PrintStream#println indirectly, via B#say.

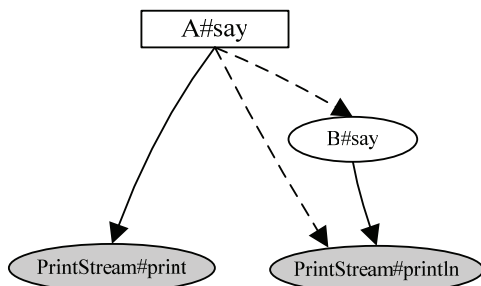


Figure 2. A multi-layered summarized instance call graph including a similar behavior set extracted from an execution trace of Fig. 1

Table 1. Classification of similar behavior

	Explanation	Model
Pair	The similar behavior consists of only two sequences.	
Vertical delegation	Three or more sequences are similar with each other and they hold delegation in transitive way.	
Horizontal delegation	Three or more sequences are similar with each other and one of them holds delegation with the others.	
Branch variation	Three or more sequences are similar each other and they have the identical class and method name.	
Clone	Other than those above	

4. Evaluation

The current implementation of tool IAnalyzer consists of three subsystems; an execution trace collector, an execution trace analyzer, and a relation visualizer. The collector, which is implemented by 1600 lines of C++ code, is a JVM agent (plug-in) to collect an execution trace from a program execution via JVMTI[9]. The analyzer, which is implemented by approximately two thousand lines of C++ code, performs the similar behavior detection from the execution trace with the algorithm presented in Sec. 3 and also classifies the detected similarities into several categories (Table 1). The visualizer is implemented by 750 lines C++ code and 141 lines of AWK script, using GraphViz[8] to draw a bitmap. The visualizer shows the user MSI-CGs of the detected similar behaviors.

To evaluate the ability IAnalyzer of the detection and the visualization of similar behavior, it was applied to an open source product batik-1.5.1[7]. The batik is a toolkit for handling a SVG (Scalable Vector Graphics) format. The toolkit contains an application program which reads a SVG file and displays it. While running this program to display a small SVG file, the execution trace collector extracted an execution trace including 520 thousands method invocations. The analyzer detected total 988 similar behavior sets from the trace, including 719 pairs, 123 vertical delegations, 4 horizontal delegations, 4 branch variations, and 138 clones.

Fig. 5 shows a MSI-CG of a similar behavior set classified into "clone". In the figure, L<digits> stands for some package or class, e.g., L16 means org.apache.batik and L13 means java.lang.String. By investigating the corresponding parts of the source code, two sequences (of methods) on the right side, ButtonFactory#createJButton and MenuFactory#createJMenuItem, perform much the same tasks but they varies in error recovery so that their code fragments do not look so similar at a glance. Also, two sequences at the center, ButtonFactory#initializeButton and MenuFactory#initializeJMenuItem, begin with the similar code but varies in the middle, as expected by the fact that they differs in the number of methods directly called by them.

5. Conclusion

This paper presented a feature of tool IAnalyzer, detection of similar behaviors from an execution of

Java program. The tool detects similar behaviors from an execution trace and categorizes the detected similarities. It also visualizes them with a diagram named MSI-CG(multi-layered summarized instance call graph). The experiment applying it to an open source product shows the ability of detecting from an execution trace of reasonable size and the usefulness of the visualization in understanding the behavior in the program.

References

- [1] H. Agrawal and J. Horgan, "Dynamic Program Slicing", *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 246-256 (1990).
- [2] E. Arisholm, L. Briand, A. Føyen, "Dynamic Coupling Measurement for Object-Oriented Software", *IEEE Transactions on Software Engineering*, vol. 30, no. 8. pp 491-506 (2004).
- [3] K. Chen, V. Rajlich, "Case Study of Feature Location Using Dependence Graph", *Proc. 8th International*

- Workshop on Program Comprehension (IWPC'00)*, pp. 241- 249 (2000).
- [4] B. Dufour, K. Driesen, L. Hendren and C. Verbrugge, "Dynamic Metrics for Java", *Proc. 9th ACM SIGPLAN Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2003)*, pp. 149-168 (2003).
- [5] M. Salah and S. Mancoridis, "A Hierarchy of Dynamic Software Views: from Object-Interactions to Feature-Interactions", *Proc. 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pp. 72-81 (2004).
- [6] M. Weiser, "Program Slicing," *IEEE Transactions on Software Engineering*, SE-10(4), pp. 352-357 (1982).
- [7] Batik SVG Toolkit, <http://xml.apache.org/batik/>
- [8] GraphViz, <http://www.graphviz.org/>
- [9] JVM Tool Interface, <http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/>
- [10] UML Resource Page, <http://www.uml.org/>

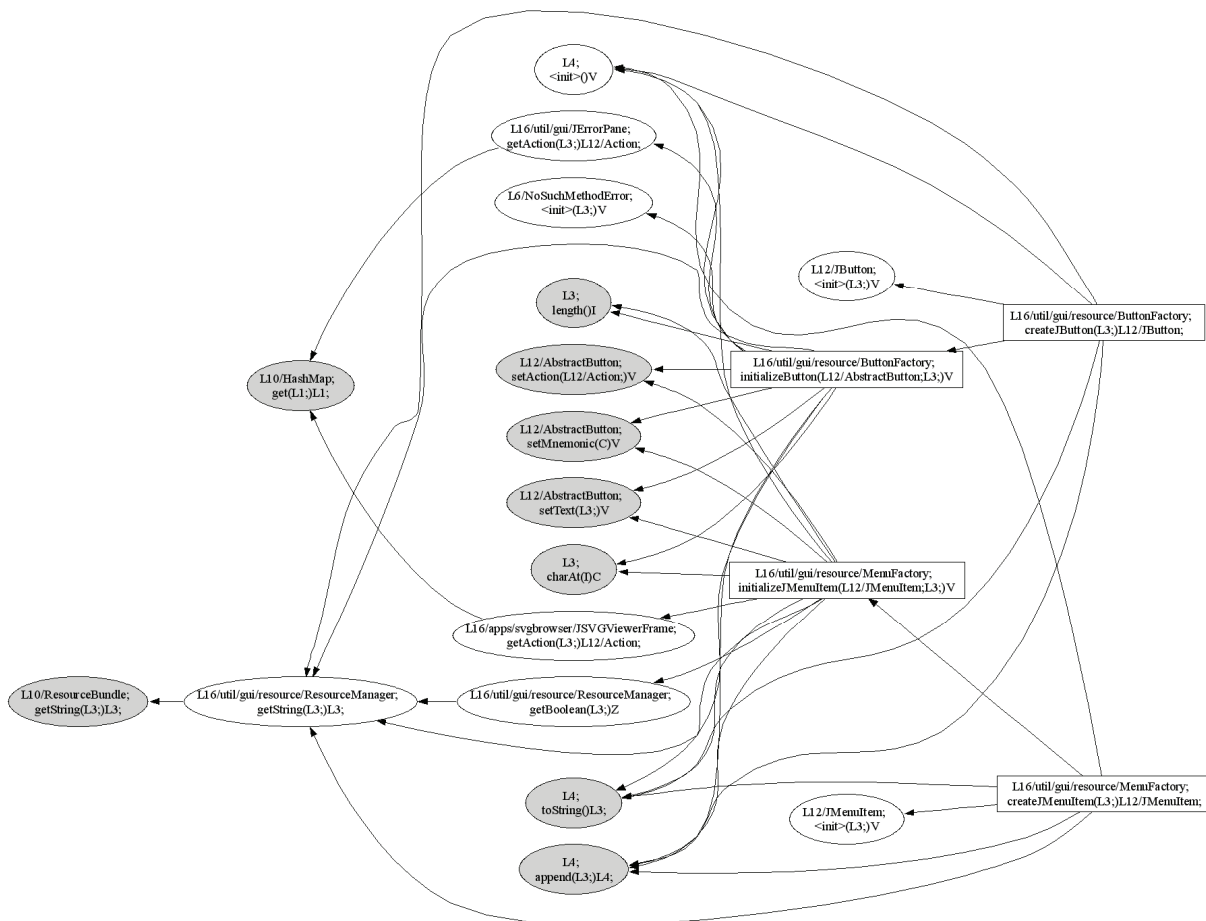


Figure 3. A MSI-CG of a similar behavior set in batik-1.5.1 detected by IIAalyzer