Extracting Sequence Diagram from Execution Trace of Java Program

Koji Taniguchi[1] Takashi Ishio[1] Toshihiro Kamiya[2] Shinji Kusumoto[1] Katsuro Inoue[1]
[1]Graduate School of Information Science and Technology, Osaka University
[2]PRESTO, Japan Science and Technology Agency
{kou-tngt,t-isio,kamiya,kusumoto,inoue}@ist.osaka-u.ac.jp

Abstract

A software system is continuously changed so many times. When we try to change a software, we must understand how the software is implemented, especially about the functions to be modified. However, such repeated changes may cause the bad situations that there is no document which reflects the changes and represents the behavior of the software correctly. So, it is important to develop a technique to extract the useful information to understand the behavior of the software from its own. We propose a method to extract compact sequence diagrams from dynamic information of object-oriented programs. Our method generates sequence diagrams by compacting a repetition included in the execution trace. This paper presents four compaction rules. The experiment illustrates how our rules effectively compact the execution trace and generate compact sequence diagrams.

1 Introduction

Generally, software is being changed by adding new functions or fixing bugs after the release. When developers try to change the software, they must understand the implementation of it. However, no documents may represent the behavior of the software correctly because the software is repeatedly changed but the documents are not updated. It is hard to understand the program behavior without appropriate documents, especially in object-oriented programs, since the dynamic behavior of the program differs from static description written in source code, owing to dynamic binding and extending[3]. It is important to develop a technique to extract the useful information for understanding of the behavior of the software from its own.

In order to take account of the interactions (message exchanges) among the objects allocated by the program, this paper proposes a method to construct compact UML[2] sequence diagrams from the execution trace of object oriented program, written in Java. Using the proposed method, the developers can easily understand the behavior of the program with generated diagrams and examine the difference between the sequence diagrams developed in design phase and produced from the program. Generally, the amount of the program execution trace tends to be very large[1]. So, it is necessary to reduce the amount of the information as possible. Our method compacts repetition parts of the execution trace. Based on the compacted execution trace, our method constructs a sequence diagram and provides it for the user (software maintainer).

We have conducted two case studies to show the usefulness of our proposed method. In the case study, our method is applied to four Java programs and their sequence diagrams are constructed. The results show the proposed compaction rules can effectively compact the execution trace. Section 2 describes the proposed method. Section 3 reports the results of the case study. Section 4 summarizes the main results.

2 Generating a Compact Sequence Diagram

2.1 Overview

Our method starts from getting an execution trace of method calls of the target program. We propose four compaction rules to reduce the amount of information since the amount of the trace is very huge. The rules compact the execution trace by abstracting some repetition patterns and recursive calls included in the trace. The compacted execution trace is translated into a compact sequence diagram. Our compaction method preserves almost the structure of the original information. Therefore, we can see which instances are interacted and how the function is implemented in the generated diagram. Since the original trace information is preserved, we also can see the uncompacted original sequences. We expect that our method is useful for understanding of the program behavior in the maintenance phase.

To explain each rule, we use a call tree which expresses a method call structure recorded in an execution trace. A node

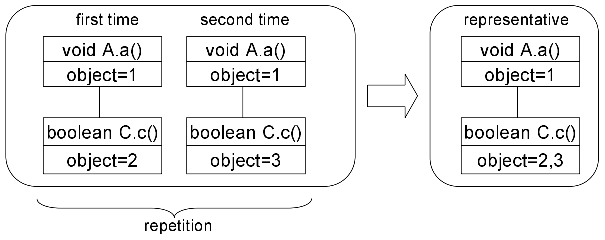

Figure 1. Making the representative by R2

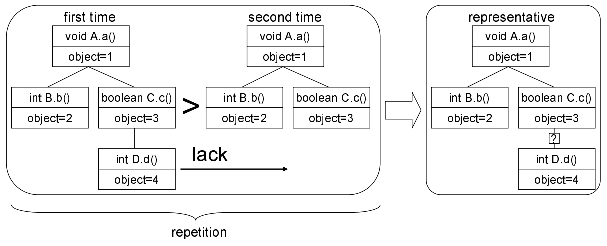

Figure 2. Making the representative by R3

of a call tree represents a method call and has the method signature and ID of the callee object.

2.2 Compaction of Repetitions

In this section, we explain the rules R1, R2 and R3 which compact the some kind of repetition patterns. They detect a repetition of *similar* subtrees in a call tree, and they replace such subtrees with one representative, which shows whole repeated structure and the number of the repetitions. The differences of each rule are which subtrees considered to be *similar* and how to make a representative.

The rule R1: Completely same

The rule R1 detects a repetition of completely same method call structure, and compacts it. In other words, R1 considers subtrees to be *similar*, if subtrees have same structure and their nodes have same methods and objects. R1 uses first one repeat unit as the representative of them. Since the representative records the number of times of the repetition, R1 does not lose information.

The rule R2: Allowing different objects

The rule R2 detects a repetition of method call structure whose object IDs may be different, and compacts it. In other words, R2 considers subtrees to be *similar*, if subtrees have same structure and their nodes have same method. R2 does not compare object IDs. To replace with the whole of the repetition, R2 makes the representative by unifying the object IDs(Figure 1).

R2 can compact subtrees which cannot be compacted by R1. However, since the result of compaction of R2 has some method calls to unified objects, it can not represent the original execution trace accurately.

The rule R3: Lack of method calls

The rule R3 detects a repetition of method call structure some of whose method calls may be *lacked* (Figure 2), and compacts it. *Lacked* means that, in comparing two subtrees, a method call which contained in one subtrees does not appear in the other one. R3 considers subtrees to be *similar*, if some method calls are *lacked* in one of the subtrees compared with the other subtrees. R3 builds the representative of the repetition based on the non-lacked subtrees (at least one non-lacked subtrees are included in the repetition), by unifying the object IDs and marking the lacked methods (Figure 2).

Since R3 can compact some repetitions whose structures may be different every one time, R3 compacts an execution trace, rather than R2. But, the result of compaction of R3 has some lacked method calls, so it does not represent the original execution trace accurately.

2.3 Compaction of Recursive Calls

In this section, we describe the rule R4 which compacts recursive call structures.

The rule R4: Recursive calls

The rule R4 detects recursive method calls, and compact the structure. We consider that the calls of the same method to different objects is also to be recursive calls.

The processes of R4 is the followings. First, we find a method which is called recursively and separate the call tree at the nodes of the recursively called method. Next, we make a set of the separated subtrees, whose elements are not *lacked* from any other subtrees which are not contained in the set. Finally, by connecting the trees contained in the set, we rebuild the call tree which shows the compact recursive call structure.

2.4 Drawing Sequence Diagram

The compacted execution trace is translated into a sequence diagram. We define annotation symbols for each compaction rule to indicate compacted part of the sequence diagram. Here, we explain the annotation symbols with examples.

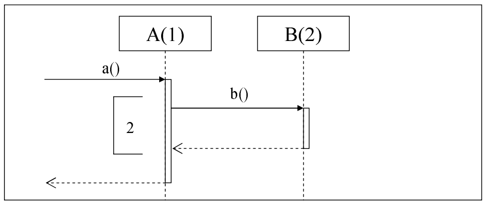

Figure 3. A generated diagram of a part compacted by the rule R1

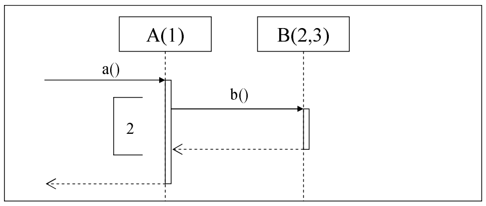

Figure 4. A generated diagram of a part compacted by the rule R2

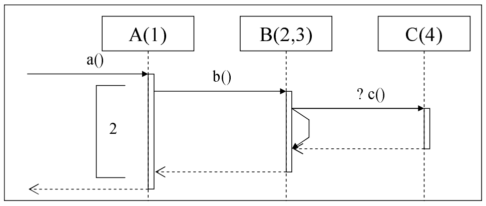

Figure 5. A generated diagram of a part compacted by the rule R3

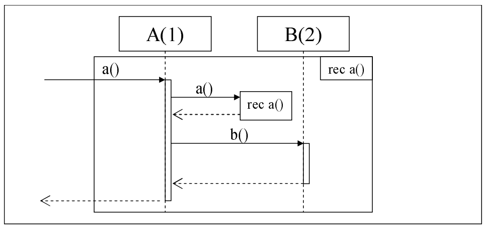

Figure 6. A generated diagram of a part compacted by the rule R4

Annotation for R1

We show the example fragment of the sequence diagram in Figure 3. In this example, the method b() of the instance B(2) is called twice during the execution of the method A(1).a(). R1 replaces the two method calls with one method call and adds an annotation symbol to the left side to indicate the repeated segment and the number of repetitions.

Annotation for R2

The example fragment is shown in Figure 4. In this example, the method b() of the instance B(2) and the same method of the instance B(3) are called in sequential order. R2 replaces the two instances with one unified object B(2, 3) and adds an annotation to the object of the diagram to indicate the unified object. R2 also adds an annotation to the left side of the repeated segment, the repetition is represented the same as R1.

Annotation for R3

The example fragment is shown in Figure 5. In this example, the instance B(2) calls the method c() of the instance C(4) and the instance B(3) does not call the method c(). R3 replaces B(2) and B(3) with the unified object B(2, 3), and R3 unifies the two executions of the method b() as one execution whose the method c() may be not called. So, R3 adds an annotation to indicate such a method. R3 also uses annotations representing repeated segments and unified objects for R1 and R2.

Annotation for R4

The example fragment is shown in Figure 6. In this example, the method a() of the instance A(1) is recursively called. R4 detects a block of the recursive calls and replaces the block with a box. The box is annotated with the name of the method recursively called, and a recursive call is represented as a method call to the block.

3 Evaluation

We evaluated the proposed method through two experiments. The first one evaluates the compaction ratio of four execution traces. Second, we try to understand a program function by using only our tool and source code.

We extracted the four execution traces and compacted them with four compaction rule. Since R4 compacts recursive calls and makes other rules effective, we applied R4 first. After applying R4, other three rules were applied from a strict compaction rule. The target programs, size of the execution traces and the results of the application of the rules are shown in Table 1. In order to evaluate the effectiveness, we define *Compaction Ratio*. We say that compaction is more effective when the compaction ratio is smaller.

$$Compaction\ Ratio\ (CR) = \frac{method\ call\ count\ after\ applying\ the\ four\ rules}{method\ call\ count\ before\ applying\ the\ rules} \times 100(\%)$$

As the result, the best compaction ratio is 0.85% for Gemini. The size is greatly reduced from original size. The

Table 1. Compaction Result

Program	Raw Data Size(# method calls)	R4	R1	R2	R3	Compaction Ratio(%)
jEdit	228764	217351	178128	16889	16510	7.22
Gemini	208360	205483	57365	1954	1762	0.85
scheduler	4398	4398	3995	238	147	3.34
LogCompactor	11994	8874	8426	208	105	0.88

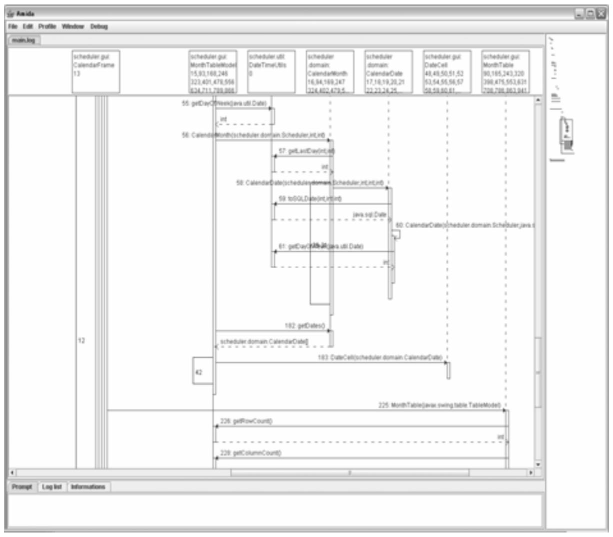
Amida
File Edit Profile Window Debug
main.log
scheduler.gui CalendarFrame 13
scheduler.gui MonthTableModel
scheduler.util DateTimeUtils 0
scheduler domain CalendarMonth
scheduler domain CalendarDate
scheduler.gui DateCell
scheduler.gui MonthTable
55: getDayOfWeek(java.util.Date)
int
56: CalendarMonth(scheduler.domain.Scheduler,int,int)
57: getLastDay(int,int)
58: CalendarDate(scheduler.domain.Scheduler,int,int,int)
59: toSQLDate(int,int,int)
java.sql.Date
60: CalendarDate(scheduler.domain.Scheduler,java.s
61: getDayOfWeek(java.util.Date)
182: getDates()
scheduler.domain.CalendarDate[]
183: DateCell(scheduler.domain.CalendarDate)
12
42
225: MonthTable(javax.swing.table.TableModel)
226: getRowCount()
228: getColumnCount()
Prompt
Log list
Informations

Figure 7. Generated Sequence Diagram from scheduler

compaction ratio of LogCompactor is 0.88%, and the actual size is 105 method calls. The size is small enough for programmers to read the meaningful information from the compacted trace. The compaction result of scheduler is 147 method calls, this is also small enough to read the execution trace. On the other hand, the compaction ratio of jEdit is 7.22%. The compacted trace includes several complex repetitions which are not handled by our four rules. In the future work, we will examine further rules to handle such repetitions. In the case of the execution trace of jEdit which has the largest number of method calls, the compaction process took about five minutes. It is practical since the compaction process is performed only once.

The diagram generated from scheduler is shown in Figure 7. A generated diagram contains the compacted repetitions identified by the proposed rules.

Second, we tried to understand how a program's function was implemented. The function which we tried to understand was the file loading function of jEdit. In this task, we used only our tool and source codes without referring to any documents and any websites. As the result of the second case study, we confirmed the effectiveness of our method. The compacted sequence diagram can be easily understood rather than the non-compacted original diagram, and the important method call information which we need to understand is not lost at all. (ex. all method calls occurred in repetitions, objects received method calls and interactions of them were indicated.) The compaction process will be performed only once per an execution trace; after that, the tool can display any parts of sequence diagram of the trace. We think that processing that time affordable in practice.

4 Conclusions

We have proposed a method to construct sequence diagrams based on the dynamic analysis (execution trace) in order to support understanding an object oriented program. We have proposed four compaction rules for execution traces to deal with the large amount of execution traces. We applied them to several Java programs. As the results, we have successfully reduced the amount of execution trace and constructed useful sequence diagrams.

There remains several future works. It is necessary to propose additional compaction rules to reduce the amount of execution trace and evaluate the sequence diagrams constructed by the proposed method. Also, we are going to deal with the programs using multi-thread.

Acknowledgements

This research was supported in part by Grant-in-Aid for Scientific Research (C)(No:17500022), JSPS.

References

[1] S. P. Reiss, M. Renieris: Encoding program executions. International Conference on Software Engineering, pp.221-230, May 2001.

[2] Unified Modeling Language (UML) 1.5 specification. OMG, March 2003.

[3] N. Wilde, R. Huitt: Maintenance Support for Object-Oriented Programs. Transactions on Software Engineering, Vol.18, No.12, pp.1038-1044, December 1992.