Sequence Diagram Slicing

Kunihiro NODA*, Takashi KOBAYASHI†, Kiyoshi AGUSA‡
Graduate School of Information Science
Nagoya University
Nagoya, Aichi, 464-8601, Japan
*Email: *knhr@agusa.i.is.nagoya-u.ac.jp,*
{†tkobaya, ‡agusa}@is.nagoya-u.ac.jp

Shinichiro YAMAMOTO
Faculty of Information Science and Technology
Aichi Prefectural University
Aichi-gun, Aichi, 480-1198, Japan
Email: yamamoto@ist.aichi-pu.ac.jp

***Abstract*—Software visualization with sequence diagrams is one of the promising techniques aimed at helping developers comprehend the behavior of object-oriented systems effectively. However, it is still difficult to understand this behavior, because the size of automatically generated sequence diagrams tends to be beyond the developer's capacity.**

In this paper, we propose a sequence diagram slicing method, which is an extension of our previous method based on a dynamic slicing technique using static information. Our proposed method is capable of accurate slice calculation based on high-precision data dependency and can support various programs, including exceptions and multithreading.

In addition, our proposed new tool performs slice calculations on the Eclipse platform and we demonstrate the applicability of this method by applying the tool to two Java programs as case studies. The results confirm the effectiveness of our proposed method for understanding the behavior of object-oriented systems.

***Keywords*-Sequence Diagram; Program Slicing; Reverse Engineering; Program Comprehension; Program Maintenance; Debugging**

I. Introduction

Understanding the behavior of a large-scale object-oriented system is one of the more difficult tasks of program comprehension. Object-oriented programs tend to contain many elements that are determined at the time of execution, because design patterns, polymorphism and delegation are often used to improve changeability and reusability of their source codes.

Visualization of interactions between objects is one of the promising techniques to help developers comprehend the behavior of object-oriented systems effectively [1] . A sequence diagram is a diagram that represents the sequence of messages passing of programs along a time line and is suitable for representing the behavior of object-oriented programs. Several tools [2], [3] support automatic generation of a sequence diagram from execution trace logs. However, it is still difficult to understand this behavior because the size of automatically generated sequence diagrams tends to be beyond the developer's capacity.

The purpose of this research is to help the developer understand the behavior of programs by providing a sequence diagram of a size that is reduced but sufficient for easy understanding.

Several methods and tools [4], [5] have been proposed to provide sequence diagrams with easy-to-understand focused behavior by reducing the amount of execution information to be visualized. We have also developed a method for providing a "partial sequence diagram" whose size is small enough to make the focused behavior of a program easy to understand by applying the program slicing technique [6]. However, in our previously proposed method, information that should be contained in a slice may be missed, because we used approximated data dependency. The method also has strong limitations when applied to programs; for example, it only supports Java 1.4 and does not cover multithread and exception occurrence programs.

In this paper we propose a sequence diagram slicing method, which extends our previous model and method, analyzes data dependency more precisely and handles the information about thread and exception to acquire the capability to be used in a wider range of applications. We also introduce our new tool "*Reticella*" that visualizes object-oriented program's behavior and supports sequence diagram slicing, which is implemented as an Eclipse plug-in. In addition, we discuss the applicability of the proposed method with two case studies.

This paper makes the following major contributions.

- We clearly define data dependency calculations for execution traces with static control structure information.
- We propose a model to represent the behavior of a multithread program that includes exception handling.
- We show the feasibility of the proposed method with an implemented tool as an Eclipse plug-in that supports all major features of Java programs.

The remainder of this paper is organized as follows. Section 2 shows our preliminary work to support comprehension of the program's behavior. We explain our proposed method in Section 3 and our implemented tool in Section 4. Section 5 shows case studies. Section 6 discusses related works and Section 7 concludes this paper.

II. Preliminary Work

We have proposed a method and a tool for generating a partial sequence diagram [7].

In our previous work, we have defined the simplified Behavior Model (B-model). The B-model represents the execution behavior of object-oriented programs and consists of execution information, method entry/exit events, caller/-callee objects of method invocation and start/end events of control structure such as a conditional branch and a loop statement. Elements of the B-model can be uniquely converted to elements of a sequence diagram.

The tool generates B-model data from source codes and execution traces, and calculates a subset of the B-model data for a partial sequence diagram by adapting our slicing method based on Dependence-Cache Slicing [8]. The proposed tool was implemented by using JavaCC [9] to statically analyze the control structure in source codes, and JDI (Java Debug Interface) in JPDA (Java Platform Debugger Architecture) to dynamically analyze how the program is being executed.

We analyze the following four kinds of dependencies in a data sequence based on the B-model that represents execution behavior.

- Control Dependency.
  A dependency that reflects control flow in programs.
- Method Invocation Dependency.
  A dependency that reflects a nested structure of method invocations.
- Start–End Dependency.
  A dependency that associates a start event with a corresponding end event in the B-model.
- Data Dependency.
  An approximate dependency between B-model data that reflects data flow in programs.

III. Proposed Method

In this paper, we propose a sequence diagram slicing method that expands our previous method mentioned in Sect. II with respect to the B-model and definitions of dependencies. The newly proposed method enables the following four types of content.

- Slice calculation based on high-precision data dependency.
- Handling programs containing exception occurrences.
- Handling multithread programs.
- Flexible choice of slicing criteria.

In the following, we explain the detail of extensions in the proposed method to achieve the abovementioned four types of content.

A. *Extension of Behavior Model*

In this section, we explain the detail of extension of the B-model. The class diagram of the extended B-model is shown in Fig. 1.

First, we describe the extension to analyze data dependency more precisely. In our previous method, the data dependency arising by assignment expression cannot be analyzed, because the B-model in our previous method does not contain information that reflects data flow related to values of variables. In the proposed method, to analyze data flow related to values of variables, we model two events, ‘definition of’ and ‘reference to’ the value of the variable during execution of the programs, and add the models to the B-model. We call these two events VariableDefinition or VariableReference, respectively.

Second, we describe the extension for covering programs that contain exception occurrences. There are two types of information about exceptions, exception occurrence and exception handling. To cover programs that contain exception occurrences, we modelled seven events, exception occurrence and start/end of try/catch/finally clause, and added these models to the B-model.

Third, we describe the extension to cover multithread programs. We need information about a thread in which an event occurs to analyze programs that contain multithreading. We modelled two details about a thread, its ID and its name, and made elements of the B-model to hold this information as attributes.

Finally, we describe the extension to choose slicing criteria more flexibly. We made elements of the B-model to hold information about variables that can be referred to at the point of the occurrence of an event as attributes. By providing information about the variables when choosing slicing criteria, it has become possible to choose slicing criteria more flexible than those chosen using our previous method.

We represent a program behavior in a data sequence $\langle b_1, b_2, \cdots, b_n \rangle$, where $b_i (1 \leq i \leq n)$ is a leaf element of the tree whose root is BehaviorEvent in Fig. 1 (e.g. MethodEntry, ConditionEnd, ExceptionOccurrence, TryStart, VariableDefinition, etc.). For any i, b_i can be converted to an element of a sequence diagram uniquely. MethodEntry and ConstructorEntry correspond to ‘a synchronous message’ and ‘a object creation message’ respectively, and both MethodExit and ConstructorExit correspond to a reply message in a sequence diagram. Other leaf elements except for VariableDefinition and VariableReference are depicted using a comment (i.e. a note symbol) in a sequence diagram. Both VariableDefinition and VariableReference are not depicted in a sequence diagram.

B. *Extension of Data Dependency*

We redefined data dependency to analyze data dependencies more precisely. In this section we explain the new definition of data dependency in the proposed method.

As mentioned earlier in Sect. III-A, we added two elements related to the value of variables to B-model, namely

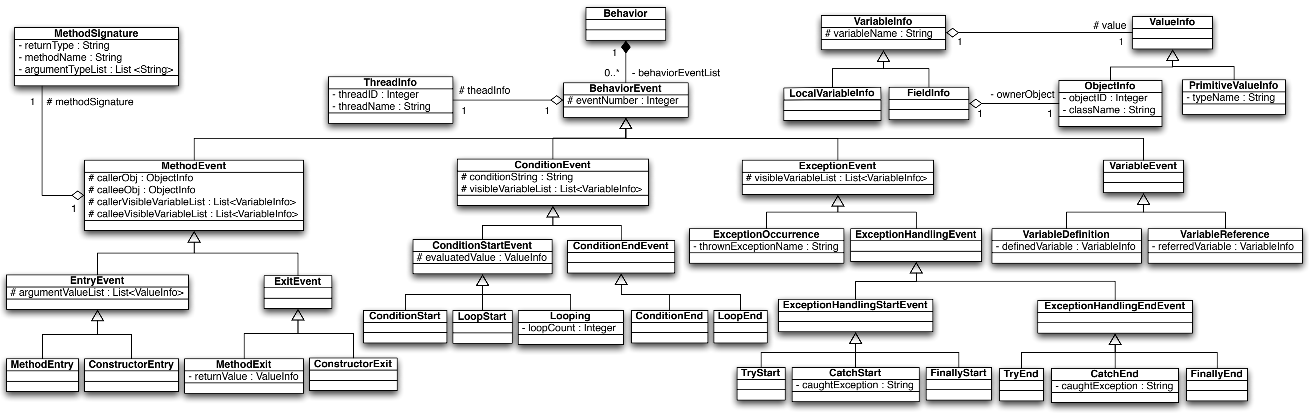


Figure 1. Class diagram of extended B-model

VariableDefinition and VariableReference. Thus, a data sequence based on the extended B-model that represents the program's behavior holds information that reflects the data flow caused by defining of or referring to values of variables. In addition, the information is sufficient for analyzing the data dependencies that arise by assignment expression that cannot be analyzed using our previous method.

In what follows, we assume that the execution behavior of a program is represented in the data sequence $\langle b_1, b_2, \cdots, b_n \rangle$, where $b_i (1 \leq i \leq n)$ is an element in the B-model, and that $1 \leq i, j \leq n$ and $i < j$ hold.

We define five kinds of data dependency to achieve high-precision data dependence analysis.

First, we define data dependency that exists from VariableDefinition to VariableReference, as follows.

Definition 1. If the following three conditions hold, there exists data dependency from b_i to b_j.

- b_i is an event of VariableDefinition of the variable v.
- b_j is an event of VariableReference of the variable v, and the definition of the value of variable v at b_i reaches the event occurrence point at b_j.

Second, we define data dependency that exists from VariableReference to another event, as follows.

Definition 2. If the following condition holds, there exists data dependency from b_i to b_j.

- Variable v is referred to at the VariableReference event b_i when executing event b_j.
  Note that one of the following conditions holds iff we say that variable v is referred to when executing an event b_j.
  - b_j is an event of method entry and the variable v is referred to as $v.method()$.
  - b_j is an event of the start of the control structure and variable v is referred to as the predicate of the event b_j.
  - b_j is the VariableDefinition of variable t and variable v is referred to when defining the value of variable t.
  - b_j is the VariableReference of variable t that is referred to by using variable v like $v.t$.
  - b_j is the VariableReference of variable t and variable t is referred to with variable v as the index of the array.
  - b_j is ExceptionOccurrence that was occurred by a throw statement and variable v is referred at the throw statement.

Finally, we define data dependencies that arise along with 'method' and 'constructor' invocations. In method and constructor invocation we should consider data flow about arguments and a return values. Note that we consider that the constructor has a return value of reference to an object, which is generated by the constructor invocation. Therefore, there are two kinds of data flow regarding a return value, that occur at the 'method exit' and 'constructor exit' events. We define three kinds of data dependencies regarding the argument and the return value of method and constructor invocation, as follows.

Definition 3. When entering 'method', formal parameters are defined by using values of the actual parameters. Thus, there exists data dependency from the VariableReference*s* of each actual parameter to the VariableDefinition*s* of each formal parameters respectively.

Definition 4. We create a variable whose name is unique during execution of the program per method invocation. When exiting from 'method' we consider that the value of the unique variable is defined by a return value. Thus, there exists data dependency from the VariableReference of the variables that affects the return value to the VariableDefinition of the unique variable. Using the return value generates the VariableReference of the unique variable.

Definition 5. We create a variable whose name is unique during execution of program per constructor invocation. When exiting from the 'constructor', the value of the unique variable is defined by reference to an object generated by the constructor invocation. Thus, there exists data dependency from the ‘constructor exit’ event (i.e. ConstructorExit) to VariableDefinition of the unique variable. Reference to the generated object generates the VariableReference of the unique variable.

C. Extension of Other Dependencies

Along with extending the B-model, we also extended the other three dependencies in our previous method. By making these extensions, it becomes possible to handle programs containing exception occurrences and multi-threading and choose slicing criteria more flexibly.

Control Dependency. In our proposed method we defined three kinds of control dependencies concerning the newly added B-model elements related to exception, as follows.

- If the following two conditions hold, there exists control dependency from b_i to b_j.
  - b_i is ConditionStart/LoopStart/Looping.
  - Decision on whether b_j is executed or not depends on the result of evaluation of the predicate of b_i.
- If the following two conditions hold, there exists control dependency from b_i to b_j.
  - b_i is TryStart/CatchStart/FinallyStart.
  - Decision on whether b_j is executed or not depends on whether b_i is generated.
- If the following two conditions hold, there exists control dependency from b_i to b_j.
  - b_i is the ExceptionOccurrence.
  - b_j is the CatchStart which catches the exception that occurred at b_i.

Method Invocation Dependency. Concerning newly added thread’s information, in our proposed method we define ‘method invocation dependency’ as follows.

- If the following three conditions hold, there exists method invocation dependency from b_i to b_j.
  - b_i is MethodEntry/ConstructorEntry.
  - b_j occurred before the exit event that corresponds to b_i occurring, or b_j is MethodExit/ConstructorExit that corresponds to b_i.
  - Both b_i and b_j are events that occurred on the same thread.

Start–End Dependency. Concerning the newly added information in relation to exception, in our proposed method we define Start–End Dependency as follows.

- If the following two conditions hold, there exists start–end dependency from b_i to b_j.
  - b_i is entry/start event.
  - b_j is exit/end event that corresponds to b_i.

D. How to Calculate a Slice of Sequence Diagram

In this section, we explain how to calculate a slice of the sequence diagram. We assume that the execution behavior of a program is represented by a B-model data sequence $\langle b_1, b_2, \cdots, b_n \rangle$ and the slicing criterion is (b_c, V), where V is a set of variables that are capable of being referred to at the point at which b_c is executed. The slice of a sequence diagram is calculated on a dependence graph we called BPDG, whose nodes are B-model elements, and its edges are based on dependencies defined in Sect. III-B and Sect. III-C. A slice calculation proceeds as follows.

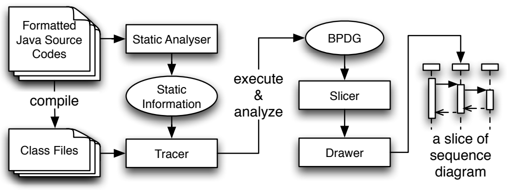


Figure 2. Overview of the tool

1) For each $v \in V$, we traverse the sequence $\langle b_1, b_2, \cdots, b_n \rangle$ backwards, that is towards the event that occurred first in the execution of the program to find VariableDefinitions $b_i^v (1 \leq i < c)$, where b_i^v represents the VariableDefinition of v and occurs ith during the program execution. Here, v and b_i^v need to satisfy either of the following two conditions.
   - v is a local variable, and the b_i^v and b_c occurred on the same thread.
   - v is a field.
2) We assume that B_{def} is a set of b_i^v for each $v \in V$. For each $b_i^v \in B_{\text{def}}$, we start slice calculation from the node b_i^v by traversing the edges backward on the BPDG.
3) The set of nodes which are reached by our traversal on the BPDG are the slice. Hence, the slice is a B-model data sequence and can be converted to a sequence diagram.
4) Finally, by converting the slice of B-model data sequence to a sequence diagram, we get a slice of a sequence diagram.

IV. Implemented tool: Reticella

In this study, we developed a tool for effective visualization of a Java program with sequence diagram, which is called *Reticella*. The developed tool is capable of calculating a slice of a sequence diagram based on the proposed method as an Eclipse plug-in. The tool consists of four parts; static analyzer, tracer, slicer and drawer. An overview of the tool is shown in Fig. 2. The developed tool calculates and generates a slice of sequence diagram automatically. However, for static analysis, before applying our tool to target programs, the source codes need to be formatted manually by the Eclipse built-in formatter. This operation is needed only once, at the first time. After this the operation needs to be repeated only if the source codes are modified. Our tool's slice calculation proceeds as follows.

1) The static analyzer analyzes source codes and acquires static information.
2) The tracer receives the static information and class files of the target program as an input and executes the programs with a specific program's input. The tracer constructs the BPDG based on the static information received from the static analyzer and the dynamic information acquired during program execution.
3) Our tool extracts from the BPDG a data sequence based on the B-model, which represents the whole behavior of programs and converts the data sequence to a sequence diagram. This is then provided to the user by the tool, using drawer.
4) The user looks at the sequence diagram, chooses a slicing criterion and inputs it into the slicer.
5) The slicer extracts a slice from the BPDG based on the slicing criterion. Then, the drawer provides it to the user as a slice of a sequence diagram.

In what follows, we explain the details of the components of our developed tool.

A. *Tracer & Slicer*

The tracer executes the program based on a specified program's input and collects information about how the program is being executed.

We implemented a tracer by using a JDI to examine how the program is being executed on Java VM. JDI is a frontend of JPDA. JDI can access a VM that is being executed and examine and control the inner states of the VM, such as receiving the notification of method entry/exit and setting the breakpoint to a specific bytecode. JDI also enables the acquisition of various pieces of information such as information on the state of a stack frame per method invocation.

The tracer captures the following events related to the B-model by using API*s* of JDI and generates B-model data along with the time line.

- Entry and exit events of method and constructor.
- Events of exception occurrence.
- Events of definition and reference of the field's value.

Other events listed above, for example start/end events of the control structure and VariableDefinition/VariableReference of local variables, cannot be captured only by JDI. To capture these events, we use the functions of setting the breakpoint at specific bytecodes in class files and notifying the event of the program counter reaching the breakpoint location, which are provided by JDI. In our tool, we set breakpoints at every statement and part of a statement in a program to know which locations in a program are executed. By doing this, we generate the B-model events that cannot be captured by JDI.

With respect to understandability, the tracer collects only information related to user codes; the tracer does not collect information related to libraries' codes.

The slicer calculates a slice by backward traversal on the BPDG using a slicing criterion inputted by the user, based on algorithm shown at Sect. III-D.

B. *Static Analyzer*

To obtain information that cannot be acquired by JDI and generate model data based on the extended B-model, we developed the static analyzer using the information of the abstract syntax tree (AST) in Eclipse JDT. The static analyzer converts source codes to AST representation and analyzes the latter.

The static analyzer analyzes mainly the Statement and Expression nodes in AST, which correspond to statement and expression in Java language respectively. The static analyzer analyzes what a data sequence based on the B-model should be generated when statement and expression are executed for all statements and expressions in a program beforehand. That is, the static analyzer makes skeletons of B-model data sequences before a program is executed. The analyzed information is passed to the tracer, which then generates a data sequence based on the B-model.

C. *Drawer*

The drawer draws a sequence diagram and provides it to the user. We used the Quick Sequence Diagram Editor [10] to draw a sequence diagram in a drawer. The Quick Sequence Diagram Editor receives text data, which is formatted based on a specific syntax as an input, then converts it to a sequence diagram and displays it. The drawer converts a data sequence based on the extended B-model to input form of Quick Sequence Diagram Editor and displays a sequence diagram by inputting the converted data into the Quick Sequence Diagram Editor. Then, the drawer provides the sequence diagram to the user.

D. *Limitation*

Unfortunately, our tool still has some limitations, as described in the following.

- Some elements in Java are not covered, for example enhanced for statement and reflection.
- Information that needs to be evaluated dynamically when the execution of a program is not analyzed, for example, the index of array and short circuit operator.
- The analysis of the tool depends on the form of bytecodes in class files and the tool cannot analyze statements that have no bytecode.

V. Case Studies

To evaluate the effectiveness of our proposed method, we applied the tool to two Java programs as case studies. In this section, we describe the case studies in detail.

```
 1 package ex1;
 2 public class Class1 {
 3     private int field;
 4     public static void main(String[] args) {
 5         Class1 obj1 = new Class1();
 6         obj1.setField(−1);
 7         try {
 8             obj1.m1();
 9             obj1.m2();
10         } catch (IllegalStateException e) {
11             e.printStackTrace();
12         }
13     }
14     public void setField(int field) {
15         this.field = field;
16     }
17     public void m1() {
18         System.out.println(this.field);
19     }
20     public void m2() throws IllegalStateException {
21         if (this.field < 0) {
22             throw new IllegalStateException();
23         } else {
24             // perform some action using positive value of field.
25         }
26     }
27 }
```

Figure 3. Source code of program 1

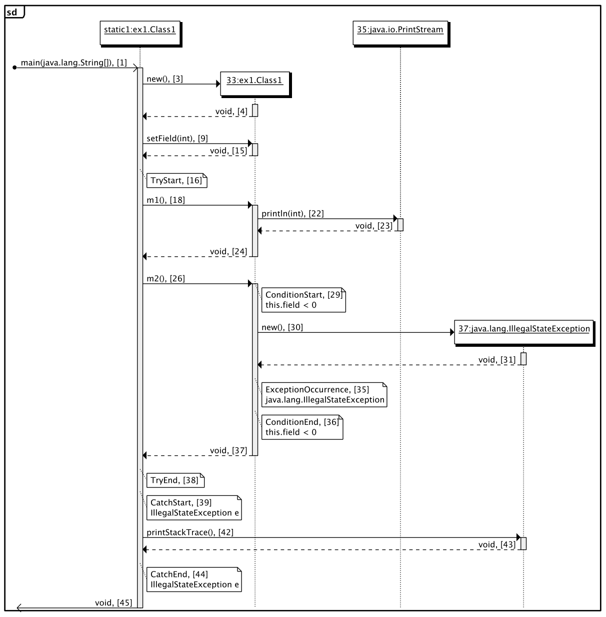


Figure 4. Sequence diagram that represent the whole execution behavior of program 1 shown at Fig. 3.

A. Case Study 1

By using our developed tool explained in Sect. IV, we visualized the program shown in Fig. 3 as a sequence diagram. As a result, we got the sequence diagram shown in Fig. 4, which represents the whole execution behavior of the program. In Fig. 4, the number between '[' and ']' indicates that the event occurred *i*th in the execution of the program. For the slicing criterion (b_c, v), the number c is chosen from numbers between '[' and ']'. In Fig. 4, we used a special lifeline whose name is "static#:FullyQualifiedClassName", where # is an arbitrary unique number per class, because a static method does not have a callee object.

In the sequence diagram shown in Fig. 4, an exception of the class "IllegalStateException" has occurred at event number [35] and the exception was caught by a catch clause at event number [39].

To detect what program behavior caused the exception, we chose (b_{42}, e) as the slicing criterion, where the variable e is able to be referred at the caller stackframe of b_{42}, and sliced the sequence diagram shown at Fig. 4.

As a result of slicing, we obtained a slice of the sequence diagram, which is shown in Fig. 5. In the program shown at Fig. 3, IllegalStateException has been thrown because: first, a negative integer value was assigned to $field$ at line 16; and second, the predicate of the if statement at line 20 was evaluated as `true`; and finally, the throw statement at line 21 was executed. The information that caused exception occurrence was indeed included in the slice shown in Fig. 5 and information irrelevant to the occurrence of exception, such as method invocation of $m1$ and $println$, was deleted and does not exist in the slice.

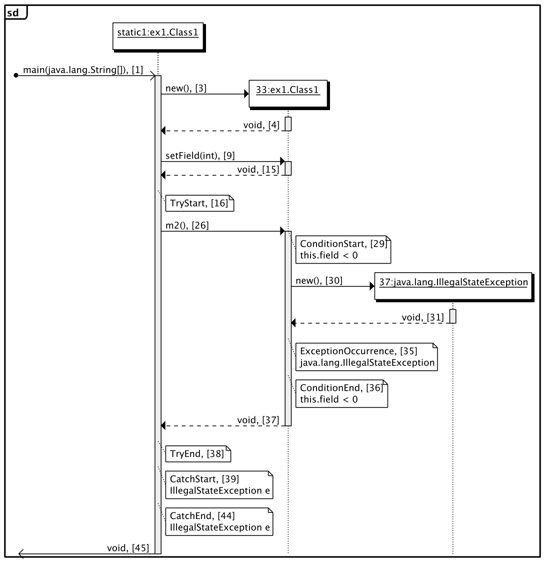


Figure 5. Slice of a sequence diagram shown at Fig. 4 by slicing criterion (b_{42}, e).

From this result, we confirmed that the data dependency arising from the assignment expression and the unusual control flow that occurs by exception occurrence—both of which could not be analyzed by our previous method—were now correctly analyzed. Moreover our proposed method is also expected to facilitate developers' tasks for large-scale programs by effectively providing useful information for understanding the program's complex behavior and time-consuming debuggings as a sequence diagram.

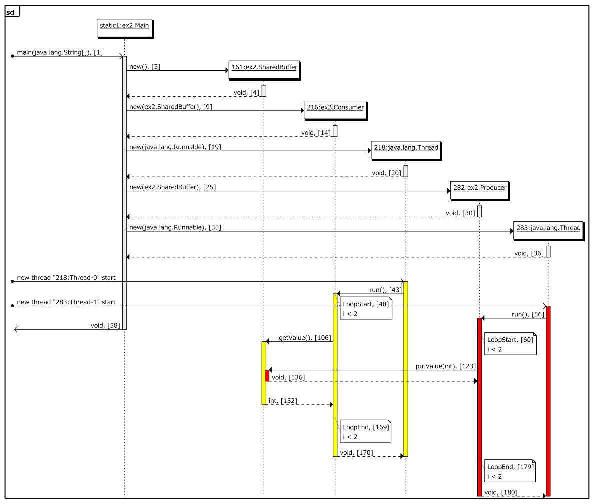


Figure 6. Slice of a sequence diagram shown at Appendix 1 by slicing criterion (b_{158}, num).

B. Case Study 2

In this section, we show an application of our proposed method to a multithread program. We chose a Consumer/Producer problem, which is typical multithreading problem, as the subject of the case study.

In the program we used in this case study, a Consumer class gets the value of common buffer by $getValue$ method and assigns the value to a variable num. Therefore, we chose (b_{158}, num) as the slicing criterion to examine what behaviors affect the value of the common buffer. Note that the event b_{158} is a method entry event for printing the value of num, after a second access of Producer/Consumer to the common buffer, and the variable num is a local variable which can be referred at the caller stackframe of b_{158}.

A generated sequence diagram that represents the whole execution behavior of the Consumer/Producer program is shown in Appendix 1. In the diagram, two sets of access to the common buffer are depicted. In the first set, the Producer stored a value to the common buffer first and the Consumer got the value from the common buffer first. In the second set, there is second access to the common buffer by the Producer and Consumer.

The result of slicing by slicing criterion (b_{158}, num) is shown in Fig. 6. The resulting diagram contains information about a Producer storing a value to the common buffer secondly and a Consumer getting the value from the common buffer secondly, and it does not contain information about the first set of accesses to the common buffer by the Consumer and Producer. Therefore, only the information related to the slicing criterion is correctly extracted in the slice.

There are many concurrent applications in recent year and because of the complexity of the behavior, comprehending the behavior of these applications is a tough task. Our proposed method has the capability to extract appropriately only information of interest that exists in several threads. Hence, our proposed method could become a valuable aid to support program comprehension and software maintenance.

VI. Related Work

A reverse-engineered sequence diagram is a very useful tool for software development such as comprehension of a program's behavior, software maintenance and debugging. There are many works and tools related to reverse-engineered sequence diagrams [11].

However, as mentioned in Sect. I, to simply generate a reverse-engineered sequence diagram from the execution trace may lead developers into difficulties, because of the large size of the diagram. To solve the problem, several works have proposed effective software visualization methods and tools.

Sharp *et al.* discuss the visual limitations and propose techniques to help developers explore large-scale reverse-engineered sequence diagrams [12]. By filtering out information of no interest, and allowing the user to select and examine details of messages on demand, effective exploration of the sequence diagram is achieved.

Taniguchi *et al.* propose a method of compressing a sequence diagram and improving its readability. [4]. If repetitions and recursive method calls exist in a program, same or similar method invocation patterns are expected to appear in a reverse-engineered sequence diagram. In Taniguchi's method, the patterns are visualized in compressed form, and thus the readability of the sequence diagram improves.

AMIDA [5] is a tool for generating sequence diagrams that represent the execution behavior of a program. AMIDA can perform automatic phase detection and separate a lot of execution information to several parts corresponding to features, and provides visualized information effectively.

JSlice [13] is one of the famous dynamic slicing tools for Java programs. It collects and analyzes bytecode traces with lossless data compression, and improves space efficiency.

VII. Conclusion and Future Work

In this study, we proposed a sequence diagram slicing method that extracts points of interest and parts related to them from a sequence diagram that represents the whole behavior of programs. The proposed method can perform slice calculation more correctly on the basis of high-precision data dependency and cover a larger variety of Java programs by extending our previous method.

We developed a tool that calculates a slice based on our proposed method. Using the tool, we applied the proposed method to two kinds of Java programs. The result confirmed that our proposed method 1) can reduce the size of information in a sequence diagram appropriately, and 2) is effective to support developers' tasks, such as comprehension of a program's behavior and debugging.

Our plans for future work and challenges are as follows.

- More reduction in the information included in a slice of a sequence diagram.
  Some approaches are expected to be effective in this challenge, for example using the method for compressing a sequence diagram together [4], introducing interactive representation of a sequence diagram like [12], and improving and refining our slicing algorithm by introducing new dependencies. Moreover, filtering out useless and unwanted information before tracing like [14] is also expected to be effective in improving the readability of the diagram and analyzing time efficiency.
- Effective representation of multithread programs on a sequence diagram.
  Depicting several threads' behavior in a program into a single sequence diagram tends to lead to more complexities and reduce the readability of the sequence diagram. Therefore, we need to introduce some new methods that cope with such problems, for example, grouping threads that are related to each other and three-dimensional representation of a sequence diagram.

ACKNOWLEDGMENT

We would like to thank Dr. Motoshi Saeki, Junya Katada and Masashi Shikauchi for their comments. This work is partially supported by a Grant-in-Aid for Scientific Research of MEXT Japan (# 20300009, # 21700027).

REFERENCES

[1] L. C. Briand, Y. Labiche, and J. Leduc, "Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software," *IEEE Trans. Softw. Eng.*, vol. 32, no. 9, pp. 642–663, 2006.

[2] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang, "Visualizing the Execution of Java Programs," in *Software Visualization (LNCS Vol.2269)*, 2002, pp. 151–162.

[3] J. Kern and C. Garrett, "Effective sequence diagram generation," Borrand White paper, 2003, http://www.borland.com/resources/en/pdf/white_papers/20263.pdf.

[4] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue, "Extracting Sequence Diagram from Execution Trace of Java Program," in *IWPSE '05*, pp. 148–154.

[5] T. Ishio, Y. Watanabe, and K. Inoue, "AMIDA: a Sequence Diagram Extraction Toolkit Supporting Automatic Phase Detection," in *Proc. ICSE Companion '08*, pp. 969–970.

[6] M. Weiser, "Program Slicing," in *ICSE '81*, pp. 439–449.

[7] J. Katada, T. Kobayashi, M. Shikauchi, and M. Saeki, "Sequence diagram generator based on slicing technique (poster)," Workshop on Eclipse Technology eXchange, 2004, http://www.se.cs.titech.ac.jp/research/sdg/index.html.en.

[8] T. Takada, F. Ohata, and K. Inoue, "Dependence-cache slicing: A program slicing method using lightweight dynamic information," in *IWPC '02*, pp. 169–177.

[9] "JavaCC," https://javacc.dev.java.net/.

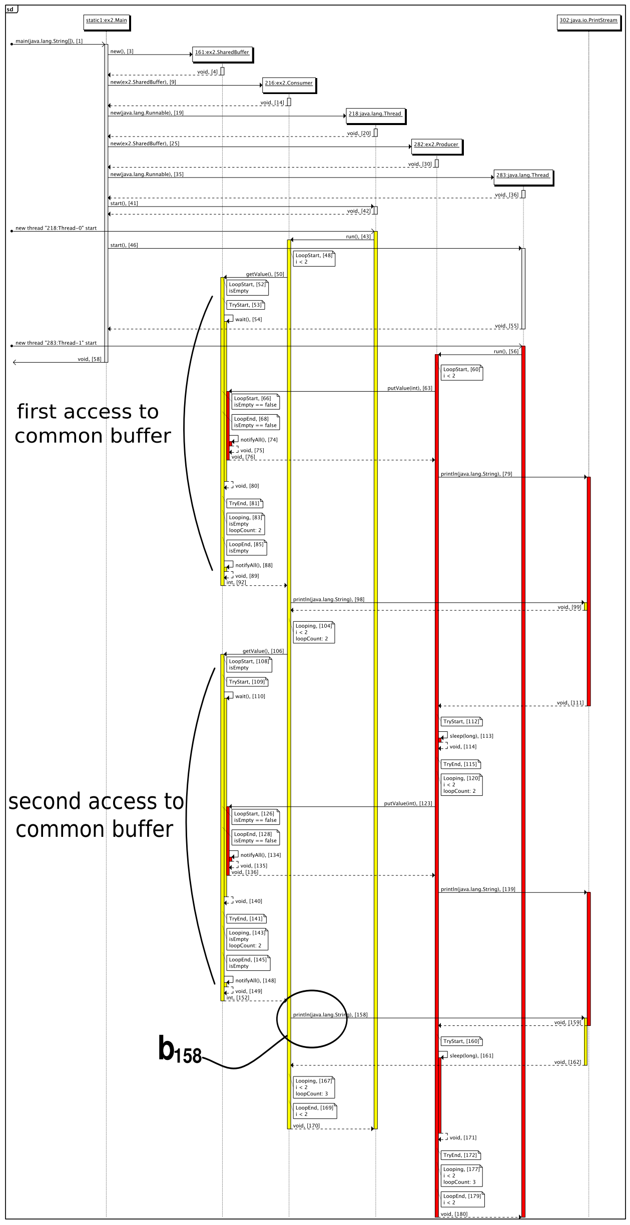
[10] M. Strauch, "Quick Sequence Diagram Editor," http://sdedit.sourceforge.net/.

[11] C. Bennett, D. Myers, M.-A. Storey, D. M. German, D. Ouellet, M. Salois, and P. Charland, "A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams," *J. Softw. Maint. Evol.*, vol. 20, no. 4, pp. 291–315, 2008.

[12] R. Sharp and A. Rountev, "Interactive exploration of uml sequence diagrams," in *VISSOFT '05*, pp. 1–6.

[13] T. Wang and A. Roychoudhury, "Using Compressed Bytecode Traces for Slicing Java Programs," in *ICSE '04*, pp. 512–521.

[14] H. Zhong, L. Zhang, and H. Mei, "Early Filtering of Polluting Method Calls for Mining Temporal Specifications," in *APSEC '08*, pp. 9–16.



Appendix 1. Sequence diagram which represents the whole execution behavior of a Consumer/Producer program.