PAPER Special Section on Knowledge-Based Software Engineering

Reticella: An Execution Trace Slicing and Visualization Tool Based on a Behavior Model

Kunihiro NODA[†], *Nonmember*, **Takashi KOBAYASHI**[†a)], **Shinichiro YAMAMOTO**[††], **Motoshi SAEKI**[†††], *and* **Kiyoshi AGUSA**[†], *Members*

SUMMARY Program comprehension using dynamic information is one of key tasks of software maintenance. Software visualization with sequence diagrams is a promising technique to help developer comprehend the behavior of object-oriented systems effectively. There are many tools that can support automatic generation of a sequence diagram from execution traces. However it is still difficult to understand the behavior because the size of automatically generated sequence diagrams from the massive amounts of execution traces tends to be beyond developer's capacity. In this paper, we propose an execution trace slicing and visualization method. Our proposed method is capable of slice calculation based on a behavior model which can treat dependencies based on static and dynamic analysis and supports for various programs including exceptions and multi-threading. We also introduce our tool that perform our proposed slice calculation on the Eclipse platform. We show the applicability of our proposed method by applying the tool to two Java programs as case studies. As a result, we confirm effectiveness of our proposed method for understanding the behavior of object-oriented systems.

key words: *program slicing, program comprehension, program maintenance, sequence diagram, reverse engineering, debugging*

1. 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 a promising technique 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 [2]. There are many tools [3]–[5] that support automatic generation of a sequence diagram from execution traces.

However, it is still difficult to understand this behavior because the size of automatically generated sequence diagrams from the massive amounts of execution traces tends to be beyond the developer's capacity.

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

Several methods and tools [6]–[9] have been proposed to provide sequence diagrams with easy-to-understand focused behavior by reducing the amount of execution information to be visualized. These methods visualize abstracted execution information by compressing control structure, or grouping and filtering visual elements.

To visualize focused behavior more properly, we focus on the program slicing technique [10]–[12] which enables us to extract a partial program related to the designated point in a program. By applying program slicing concept to the sequence diagram, we provide "*slice*" of a sequence diagram whose size is small enough to be easy to understand focused behavior of a program.

In this paper we propose a sequence diagram slicing approach [13], [14] based on a execution trace slicing and visualization method. Our proposed method is capable of slice calculation based on a behavior model which can treat dependencies based on static and dynamic analysis and supports for various programs including exceptions and multi-threading. We also introduce our object-oriented program behavior visualization tool "*Reticella*" supporting sequence diagram slicing implemented as an Eclipse plug-in. In addition, we discuss applicability of 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 B-model which is a model to represent the behavior of a multi-thread 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 discusses related works. We explain our proposed method in Sect. 3 and our implemented tool in Sect. 4. Sec-

Manuscript received July 4, 2011.
Manuscript revised October 31, 2011.

[†]The authors are with the Department of Information Engineering, Graduate School of Information Science, Nagoya University, Nagoya-shi, 464–8601 Japan.

[††]The author is with the Faculty of Information Science and Technology, Aichi Prefectural University, Nagakute-shi, 480–1198 Japan.

[†††]The author is with the Department of Computer Science, Graduate School of Information Science and Engineering, Tokyo Institute of Technology, Tokyo, 152–8552 Japan.

a) E-mail: tkobaya@acm.org

DOI: 10.1587/transinf.E95.D.959

tion 5 shows case studies and Section 6 concludes this paper.

2. 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 [5]. As mentioned in Sect. 1, to simply generate a reverse-engineered sequence diagram from the execution trace may lead developers into difficulties, because of the large amount of information included in the execution trace. To solve the problem, there are many works to handle the huge amount of information and visualize the software behavior effectively.

Jinsight [3] is a tool that visualizes the run-time behavior of a program. Jinsight collecting only the specified runtime information, reduces the amount of size of information to be visualized. Moreover, Jinsight can filter out the information in which user is not interested, for example hide the information about particular classes and visualize only call history leading to specific method invocation.

Sharp *et al.* discuss the visual limitations and propose techniques to help developers explore large-scale reverse-engineered sequence diagrams [15]. 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.

The difference point between our method and these filtering approaches is the way to reduce information. Our method, due to use of program slicing technique, reduces the information based on data and control dependencies, while these filter approaches reduce the information based on the query input by user or the call stack depth. Because we adopt program slicing technique, our method is more effective in software debugging tasks and software maintenance tasks.

Taniguchi *et al.* propose a method of compressing a sequence diagram and improving its readability [6]. 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.

Bohnet *et al.* prunes information, which is less relevant for programmer, from massive execution sequence [9]. In addition, they detect repetitive sections in the pruned trace and visualize the sections as compact forms. Applying their method to very large scale trace, they evaluated the effectiveness of the method.

Bennett *et al.* discuss about the features providing cognitive support for comprehending the large-scale reverse engineered sequence diagrams, such as diagram's layout, navigation and abstraction like grouping. They developed the effective sequence diagram viewer based on the discussed features [5].

AMIDA [7] is a tool for generating sequence diagrams that represents 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.

In our proposed method, we use program slicing technique to reduce the irrelevant information. Program slicing is widely researched and there exists many variation of slicing technique [16], [17].

In dynamic slicing, it is often that the cost of analysis becomes a major problem. Coping with the problem, several cost effective and memory efficient slicing techniques are proposed [12], [18]. JSlice [18] is a famous dynamic slicing tool for Java programs. It collects and analyzes bytecode traces with lossless data compression, and improves space efficiency.

3. Proposed Method

3.1 Outline

We propose a method and a tool for "sequence diagram slicing" which extracts a partial sequence diagram related to user specified event of execution traces [13], [14]. A partial sequence diagram is a part of a sequence diagram that represents the whole execution behavior of a program.

In this paper, we define the simplified behavior model named "B-model" and dependencies on the B-model. The B-model represents the execution behavior of object-oriented programs. Our proposed tool generates a dependence graph "BPDG" whose nodes are B-model elements which represents the whole of a program's behavior from source codes and execution traces, and calculates a subset of the B-model data by adapting our slicing method based on Dependence-Cache Slicing [19]. The subset of the B-model data is to be converted to a partial sequence diagram.

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

- Data Dependency.
  An approximate dependency between B-model data that reflects data flow in programs.
- 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.

3.2 B-model: Simplified Behavior Model

B-model is a simplified behavior model for object oriented programs. We focus a subset of behavior information related to visualization with sequence diagrams. The class diagram of the B-model is shown in Fig. 1. B-model consists of ex-

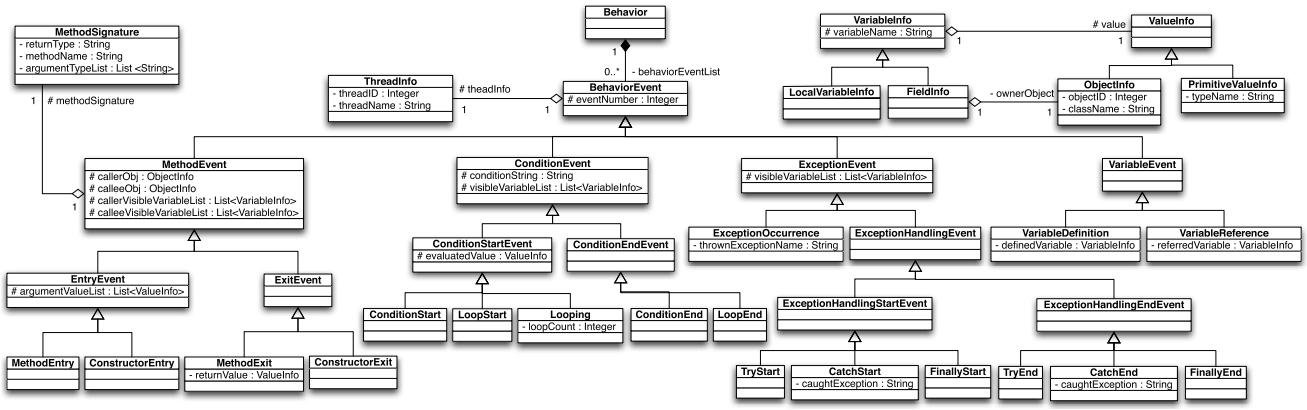

Fig. 1 Class diagram of B-model.

ecution 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, and events related to exception handling such as an occurrence of an exception and start/end events of try-catch clauses. Elements of the B-model can be uniquely converted to elements of a sequence diagram.

We represent a program behavior in an event 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 are depicted as 'a synchronous message' and 'an object creation message' in a sequence diagram respectively, and both MethodExit and ConstructorExit are depicted as 'a reply message' in a sequence diagram. Leaf elements of ConditionEvent and ExceptionEvent such as ConditionStart, Looping, ExceptionOccurrence and TryEnd are depicted using a comment (a note symbol) in a sequence diagram. Note that Looping is an event representing beginning next iteration of the loop and ExceptionOccurrence represent that an exception is thrown. Both VariableDefinition and VariableReference are not depicted in a sequence diagram.

3.3 Dependencies on B-model

3.3.1 Data Dependency

As mentioned in Sect. 3.2, we introduce two elements related to the value of variables to B-model, namely VariableDefinition and VariableReference. Thus, an event sequence based on the 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.

In what follows, we assume that the execution behavior of a program is represented in the event 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 six kinds of data dependencies for data dependence analysis.

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

Definition 1. If the following two 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 instance of MethodEntry for method *method1* and the variable v is referred to as $v.method1()$.
  - 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.

Third, 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 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 return value, that occur at the 'method exit' and 'constructor exit' events. We define three kinds of data dependencies regarding the arguments and the return values of method and constructor invocations, 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 parameter 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.

If a cascading method invocation such as $m1().m2()$ exists, we virtually treat the code as the form $\_obj1 = m1(); \_obj1.m2()$, where $\_obj1$ is a temporarily unique variable, and then calculate dependencies for the cascading method invocation.

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.

Finally, we define an approximate data dependency. When analyzing the behavior of library methods, it's highly likely that its source codes do not exist. In the case, complete analysis of the behavior occurred in library methods is very difficult. To deal with the case, we define the approximate data dependency as follows.

Definition 6. If the following conditions hold, there exists approximated data dependency from b_i to b_j.

- The relation between object's identification obj_id1, obj_id2 and event b_i is one of the following.
  - b_i is an instance of MethodEntry whose callee object's id is obj_id1, and the arguments of the MethodEntry b_i contain the object whose identification is obj_id2.
  - b_i is an instance of MethodExit whose caller object's identification is obj_id1, and the return value for an instance of MethodExit b_i is the object whose identification is obj_id2.
  - b_i is an event of constructor entry whose caller object's identification is obj_id1, and the identification of the object created by the constructor is obj_id2.
- b_j is an event occurred in a method whose callee object's identification is obj_id1, and the object whose identification is obj_id2 is referred at the event b_j. Note that the relation of an event and referred objects is shown in Table 1.

Table 1 The relation of an event and referred objects.

Event name	Referred objects
MethodEntry, ConstructorEntry	callee object, objects as arguments
MethodExit	object as the return value
ConditionStart, LoopStart, Looping	objects referred

By this definition, data dependencies existing between methods defined by user and library methods are analyzed approximately at a method invocation level.

3.3.2 Other Dependencies

Control Dependency. In our proposed method, we define three kinds of control dependencies concerning 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 thread's information, 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 information related to exception, 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.

3.4 Calculating a Slice of Sequence Diagram

The slice of a sequence diagram is calculated on BPDG which consists of B-model data and dependencies defined in Sects. 3.3 and 3.3.2. We assume that the execution behavior of a program is represented by an event sequence $\langle b_1, b_2, \cdots, b_n \rangle$ and the slicing criterion is (b_c, V), where b_c is an arbitrary event in the event sequence and V is a set of variables that are capable of being referred to at the point at which b_c is executed. A slice calculation proceeds as follows.

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 an event sequence and can be converted to a sequence diagram.
4. Finally, by converting the slice of event sequence to a sequence diagram, we get a slice of a sequence diagram.

3.5 Relationship with Dynamic Program Slicing

Since we calculate a slice of an execution trace based on dependencies between events which are results of dynamic and static program analysis, we can theoretically obtain the same event sequence if we perform the following process with all source codes of target software including standard API libraries:

1. Apply a dynamic program slicing technique to target software and obtain a slice which is an executable subset of target software such as the set of statements.
2. Execute the slice with a tracer and record events.

In the case that we cannot access all source codes, the approach that needs information of source codes might not be applicable. Our proposed method can calculate a slice even if a part of source codes is missing by using the approximate data dependency. Bytecode base approaches such as JSlice [18] allow us to calculate an executable slice without source codes. However approaches based on bytecode strongly depend on a language.

The first advantage of our approach is the execution cost. Our proposed method focused on the visualization of a behavior with a sequence diagram and records the behavior information which is represented with a sequence diagram only. Once the behavior information is record as BPDG by using proposed B-model, we can re-calculate and re-render a slice of a sequence diagram without any execution of target software.

The second advantage is portability of our approach. Our slicing and visualization method are language-independent because it is based on B-model just only. We can easily apply our approach to other object-oriented languages by implementing only a tracer which records a behavior as B-model.

4. 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 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.

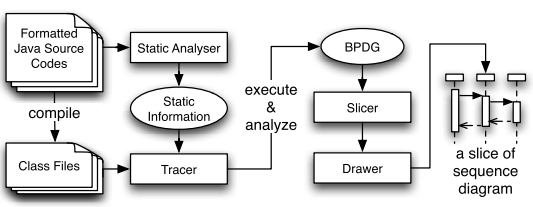

Fig. 2 Overview of the tool.

3. Our tool extracts an event sequence based on the B-model, which represents the whole behavior of programs and converts the event sequence to a sequence diagram, from the BPDG. 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.

4.1 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 which are not 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 parts 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.

The tracer collects only information related to user codes; the tracer does not collect information about computations with libraries. Our tracer automatically recognize whether a class is belonging to user codes or not by checking the existence of source codes for the class in input files. Classes with no source codes are recognized as libraries which are not target to visualize the behavior.

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. 3.4.

4.2 Static Analyzer

To obtain information that cannot be acquired by JDI and generate model data based on the 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 an event 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 event sequences before a program is executed. The analyzed information is passed to the tracer, which then generates an event sequence based on the B-model.

4.3 Drawer

The drawer draws a sequence diagram and provides it to the user. We used the Quick Sequence Diagram Editor [20] 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 an event sequence based on the 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.

4.4 Limitation

Our tool still has some limitations. Our tool covers any Java 6 program which does not contain the following elements.

- enhanced for statement
- reflection
- native method call
- conditional operator (&&, | |, and ? :)
- statement that have no bytecode (e.g. empty while loop)

In addition, because our tool does not analyze the value of an index of an array variable, the size of a slice is relatively greater than that of conventional dynamic slicing.

5. 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.

5.1 Case Study 1

In this section, we show an application of our proposed method to a multi-thread program. The program of Consumer/Producer problem, that we use in this case study, has three key classes, `Consumer`, `Producer`, and `SharedBuffer`. `SharedBuffer` class has a private field named *value*, which is a common buffer used between an object of `Consumer` class and an object of `Producer` class. The object of `Consumer` class gets a value from the common buffer by the method *getValue* and the object of `Producer` class stores a value to the common buffer by the method *putValue*. The objects of `Consumer` and `Producer` work in different threads and try to get/put a value from/to the common buffer.

A generated sequence diagram that represents the whole execution behavior of the Consumer/Producer program is shown in Fig. A· 1. In Fig. A· 1, the number between '[' and ']' indicates that the event occurred ith in the execution of the program. For the slicing criterion (b_c, v), the number c is chosen from numbers between '[' and ']'. In Fig. A· 1, 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 diagram, two sets of accesses 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 are second accesses to the common buffer by the Producer and Consumer.

We chosed (b_{158}, num) as the slicing criterion to examine what behavior affects the value of the common buffer, where *num* is a local variable of an object of `Consumer` class and the variable *num* holds the return value of the *getValue* method. Because the value of *num* reflects the value of the common buffer, extracting a slice affecting the value of *num* enables us to examine what behavior affects the value of the common buffer. Note that the event b_{158} is a MethodEntry event for printing the value of *num*, after a second access of Producer/Consumer to the common buffer.

The result of slicing by slicing criterion (b_{158}, num) is shown in Fig. 3. 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 into 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

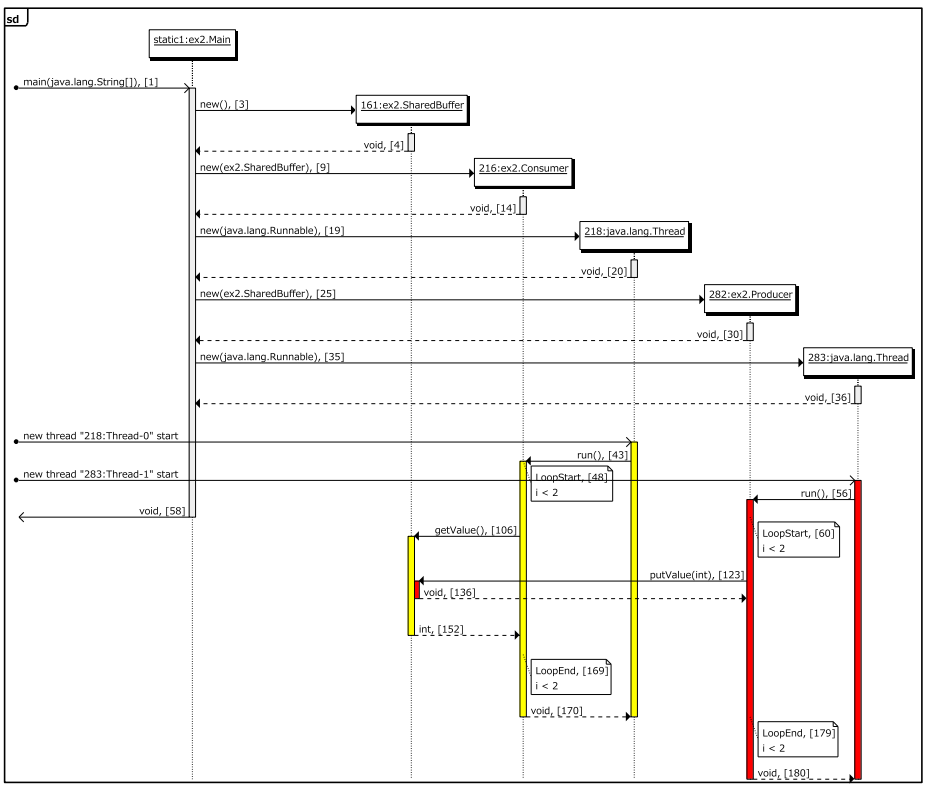

Fig. 3 Slice of a sequence diagram shown at Fig. A· 1 by slicing criterion (b_{158}, num).

support program comprehension and software maintenance.

Our approach only partly supports to comprehend the complexity of concurrent application because B-model does not have information related to mutual exclusion and execution time. We can support to comprehend more aspects of concurrent application with extensions of B-model to treat the information related to mutual exclusion which can be obtained from Java VM using JDI.

5.2 Case Study 2

In this section, we apply our proposed method to a more bigger program: a simple editor program that is implemented by using Java Swing classes and has basic functions, such as changing font style, font color, font size and font family. By using *Reticella*, we analyzed the editor's behavior with following operations:

1. Type some words.
2. Change font family and font style.
3. Type some words.

A generated sequence diagram shown in Fig. 4 represents the whole execution behavior of the editor program. The sequence diagram shown in Fig. 4 contains 1490 messages (4217 B-model events). This information is too huge to understand the behavior. Even though we did few operations, too many messages are contained in the sequence diagram because methods for obtaining and updating current attributes of font's style are frequently invoked every time we update the caret position and type a character.

To examine the behavior of changing font family, we chose $(b_{4087}, currentFontFamily)$ as a slicing criterion, where b_{4087} is an event after second operation mentioned above and the variable *currentFontFamily* is a variable that contains the information about current font family. The result of slicing is shown in Fig. 5. The sequence diagram shown in Fig. 5 contains only 24 messages (58 B-model events), and the diagram only has information about method invocations that affect the value of the slicing criterion, such as *getAttribute* method and *getFontFamily* method invocation.

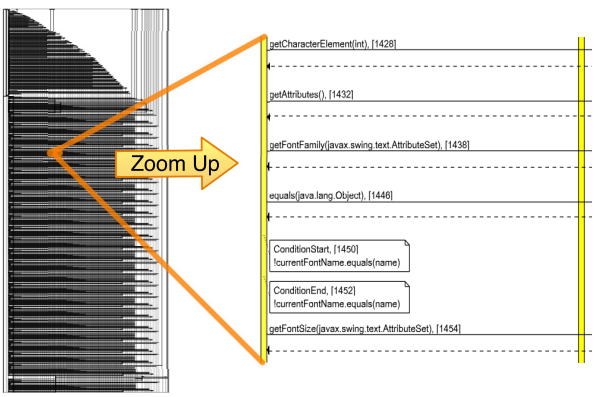

Fig. 4 Sequence diagram which represents the whole execution behavior of the editor program.

The reduction rate of messages is 1.61%[†] (The reduction rate of B-model events is 1.37%[††]) and the size of the sliced sequence diagram is enough small to understand the focused behavior. Thus, it is confirmed that our proposed method can handle large volumes of information, and extract desired information related to focused behavior from it appropriately.

5.3 Discussion

From two case studies, we confirm that our proposed method has the capability of reducing the huge amount of information in execution trace to the size that is sufficient for easy understandings. In our proposed method, we use the approximate data dependency mentioned in Sect. 3.3 when analyzing data dependencies between library codes and user codes. Because of approximation of the data dependency, several elements that should be contained in the resulted slice may be missed. However, from the perspective of a sequence diagram, we can obtain a good result by only analyzing data dependencies at method invocation level because a sequence diagram mainly consist of message sequences; detailed events, such as variable definition or reference events, do not be depicted in a sequence diagram. In addition, due to the approximation of the data dependency, the cost of analyzing dependencies in library codes is reduced in our method while it is highly expensive to analyze all of information in library source codes. Thus, our proposed method is efficient and effective way to support comprehension of the focused behavior and software maintenance tasks.

6. Conclusion and Future Work

In this study, we proposed an execution trace slicing and visualization method based on a behavior model. Our proposed sequence diagram slicing method provides partial sequence diagrams related to points of developer's interests from a sequence diagram that represents the whole behavior of programs. Our proposed method is capable of slice calculation based on a proposed behavior model which can treat dependencies based on static and dynamic analysis and supports major features of Java languages including exceptions and multi-threading.

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 fol-

[†]A reduction rate of messages is calculated by the formula: (the number of messages in a slice) / (the number of messages in an execution trace) × 100.

[††]A reduction rate of B-model events is calculated by the formula: (the number of B-model events in a slice) / (the number of B-model events in an execution trace) × 100.

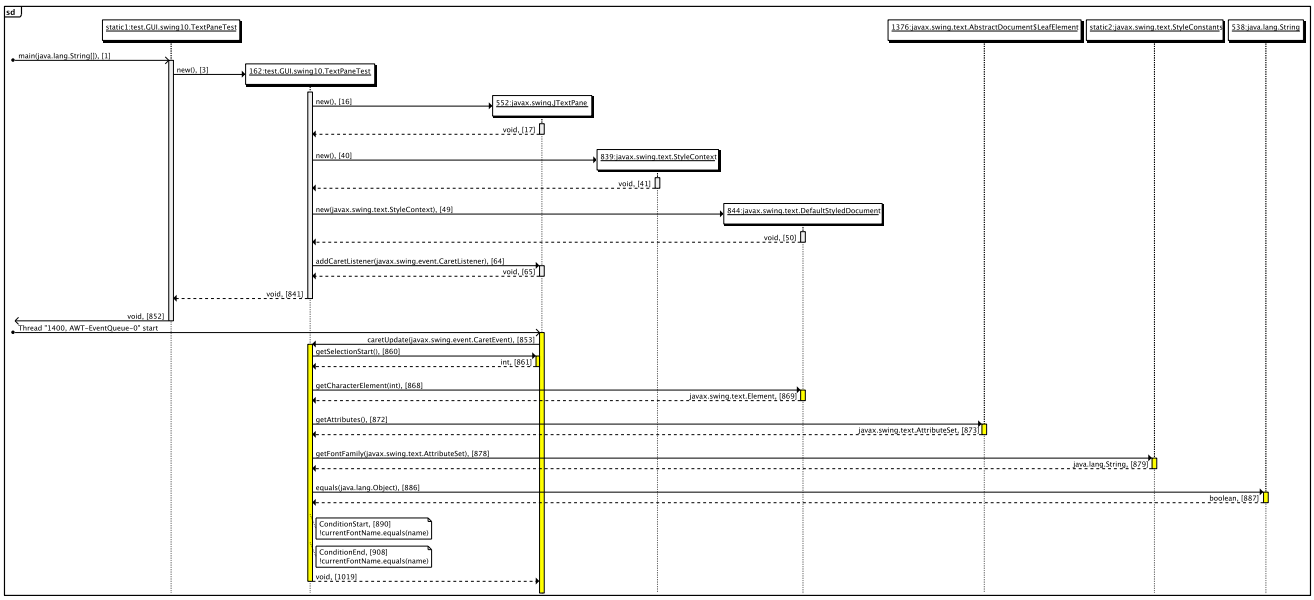

Fig. 5 The slice of the sequence diagram shown in Fig. 4 by the slicing criterion (b_{4087}, *currentFontFamily*).

lows.

- 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 [6] together, introducing interactive representation of a sequence diagram like [15], and improving and refining our slicing algorithm by introducing new dependencies. Moreover, filtering out useless and unwanted information before tracing like [21] is also expected to be effective in improving the readability of the diagram and time-efficient analysis.
- 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 Junya Katada and Masashi Shikauchi for their comments and their prototype of B-model and slicing tools. This work is partially supported by a Grant-in-Aid for Scientific Research of MEXT Japan (# 20300009, # 21700027).

References

[1] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J.J. van Wijk, "Execution trace analysis through massive sequence and circular bundle views," J. Syst. Softw., vol.81, no.12, pp.2252–2268, 2008.
[2] 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.
[3] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J.M. Vlissides, and J. Yang, "Visualizing the execution of Java programs," Software Visualization (LNCS vol.2269), pp.151–162, 2002.
[4] J. Kern and C. Garrett, "Effective sequence diagram generation," Borland White paper, 2003. http://www.borland.com/resources/en/pdf/white_papers/20263.pdf
[5] 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.
[6] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue, "Extracting sequence diagram from execution trace of Java program," IWPSE '05, pp.148–154, 2005.
[7] T. Ishio, Y. Watanabe, and K. Inoue, "AMIDA: A sequence diagram extraction toolkit supporting automatic phase detection," Proc. ICSE Companion '08, pp.969–970, 2008.
[8] C. Bennett, D. Myers, M.-A. Storey, and D. German, "Working with 'Monster' traces: Building a scalable, usable sequence viewer," Proc. 3rd International Workshop on Program Comprehension through Dynamic Analysis (PCODA '07), pp.1–5, 2007.
[9] J. Bohnet, M. Koeleman, and J. Doellner, "Visualizing massively pruned execution traces to facilitate trace exploration," VISSOFT '09, pp.57–64, Sept. 2009.
[10] M. Weiser, "Program slicing," ICSE '81, pp.439–449, 1981.
[11] H. Agrawal and J.R. Horgan, "Dynamic program slicing," PLDI '90, pp.246–256, 1990.
[12] X. Zhang and R. Gupta, "Cost effective dynamic program slicing," PLDI '04: Proc. ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, pp.94–106, 2004.
[13] 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
[14] K. Noda, T. Kobayashi, K. Agusa, and S. Yamamoto, "Sequence

Appendix

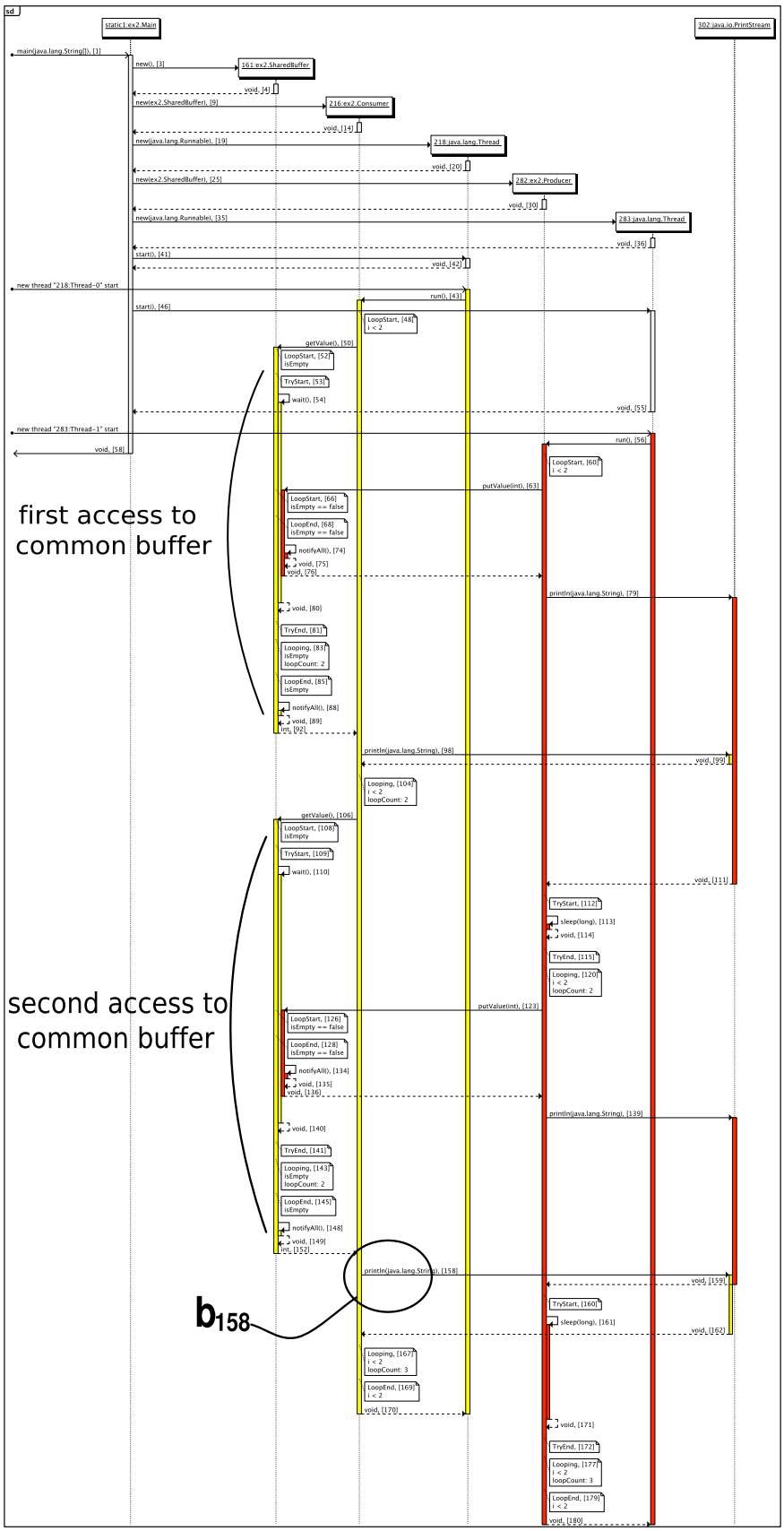

Fig. A· 1 Sequence diagram which represents the whole execution behavior of a Consumer/Producer program.

diagram slicing," APSEC '09, pp.291–298, 2009.
[15] R. Sharp and A. Rountev, "Interactive exploration of UML sequence diagrams," VISSOFT '05, pp.1–6, 2005.
[16] F. Tip, "A survey of program slicing techniques," J. Programming Languages, pp.3:121–189, 1995.
[17] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," SIGSOFT Softw. Eng. Notes, vol.30, no.2, pp.1–36, 2005.
[18] T. Wang and A. Roychoudhury, "Using compressed bytecode traces for slicing Java programs," ICSE '04, pp.512–521, 2004.
[19] T. Takada, F. Ohata, and K. Inoue, "Dependence-cache slicing: A program slicing method using lightweight dynamic information," IWPC '02, pp.169–177, 2002.
[20] M. Strauch, "Quick sequence diagram editor," http://sdedit.sourceforge.net/
[21] H. Zhong, L. Zhang, and H. Mei, "Early filtering of polluting method calls for mining temporal specifications," APSEC '08, pp.9–16, 2008.

Kunihiro Noda received a B.S. and an M.S. in computer science from Nagoya University in 2009 and 2011 respectively. He currently works as a software engineer. His research interests include software comprehension and program analysis.

Takashi Kobayashi received a B.Eng., M.Eng. and Dr.Eng. degrees in computer science from Tokyo Institute of Technology in 1997, 1999 and 2004 respectively. He is currently an associate professor of Graduate School of Information Science, Nagoya University. His research interests include software reuse and software comprehension and multimedia information retrieval and data mining.

Shinichiro Yamamoto received a B.S. in electrical & electronic engineering, and an M.S. and a Ph.D. in computer science from Nagoya University in 1985, 1987 and 1995 respectively. He is currently a professor of school of information science and technology at Aichi Prefectural University. His research interests include software analysis, software visualization and software comprehension.

Motoshi Saeki received a B.S. in electrical & electronic engineering, and an M.S. and a Ph.D. in computer science from Tokyo Institute of Technology, in 1978, 1980 and 1983 respectively. He is currently a professor of computer science at Tokyo Institute of Technology. His research interests include requirements engineering, software design methods, software process modeling, and computer supported cooperative work (CSCW).

Kiyoshi Agusa received a B.S. and M.S. in electrical engineering, and a Ph.D. in computer science from Kyoto University in 1970, 1972 and 1980 respectively. He is a professor of Graduate School of Information Science, Nagoya University. His roles in Nagoya University are a director of Information Technology Center, and a vice director of Information and Communication Headquarters. His research interests are related to software engineering, especially programming environments, requirements engineering, CASE.