The Omission Finder for Debugging What-Should-Have-Happened Bugs in Object-Oriented Programs

Kouhei Sakurai
Kanazawa University
Ishikawa, Japan
k_sakurai@acm.org

Hidehiko Masuhara
Tokyo Institute of Technology
Tokyo, Japan
masuhara@acm.org

ABSTRACT

Trace-based debuggers help a debugging process by displaying a history of executed operations with their parameters in a run of a program. However, those debuggers are unable to provide any clues when a program does *not* perform operations that ought to occur. This paper proposes a novel feature called the *omission finder* for trace-based debuggers. This feature correlates points-to analysis results with an execution history to show operations that could have been but actually were not performed on a specified instance if the program behaved differently. We implemented the omission finder on top of an existing trace-based debugger, and confirmed reduction of the number of debugging steps with an omission bug in a real-world program. Our user-study also showed reduction of debugging times with programs containing omission bugs.

Categories and Subject Descriptors

D2.5 [**Software Engineering**]: Testing and Debugging—Debugging aids, Tracing; D3.4 [**Programming Languages**]: Processors—Debuggers

General Terms

Algorithms, Languages, Design, Performance

Keywords

Omission errors, Trace-based debugger, Points-to analysis

1. INTRODUCTION

Among the techniques for building software systems—including code reviewing, verification, software testing, and debugging—debugging is an inevitable process of finding a code fragment in a program that needs to be fixed.

Before explaining the background of the paper, we define a few terms relevant to debugging. As shown in Figure 1(a), a *defect* is an incorrect code fragment in a program. When running a program, the execution of a defect produces an incorrect program state, or an *error*. Errors can be *propagated*; i.e., execution with an error often results in another error. Finally, errors are *observed* in outputs from a program. We informally refer to a defect as a *bug*, especially in a situation before identifying its location in a program.

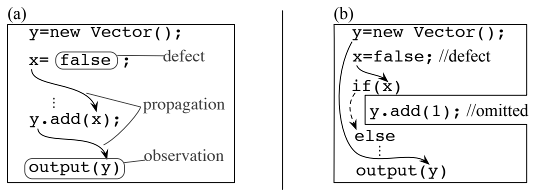


Figure 1: (a) A defect in a program (top) yields an error (in x), that propagates to another error (in y) eventually observed (bottom). (b) An error (in x) caused by a defect (top) lets a conditional jump incorrectly omit a branch, which causes another error—an execution omission error—(in y) by not executing a statement that ought to be performed.

A debugging process usually starts from a code fragment that produced an observed error, and chases after causes of errors until we encounter a defect. When an error is observed after many propagation steps, we need to traverse a lengthy path in a program.

In this paper, we present an extension of the *trace-based debugger* by adding a novel feature called the *omission finder* to handle *execution omission errors*. A trace-based debugger, also known as a back-in-time or omniscient debugger, first runs a program to collect information on executed operations (what we call an *execution history*) and then allows the developer (i.e., the user of the debugger) to inspect the states of a program, such as values in variables and instance fields, at any point of execution [3, 4, 5, 10, 12, 11]. One of the advantages of these debuggers is the direct support of backward navigation: starting from a certain point in an execution, it can go back to an earlier point and show that state. This is in contrast to break-point debuggers, which can only move forward and require the developer to re-execute a program from the beginning in order to inspect states at an earlier execution point.

In this paper, we deal with *an execution omission error*, which is an error caused by the omission of execution of a branch[1] that ought to be executed. For example, the program in Figure 1(b) incorrectly sets the value of `x` to

[1]In the paper, we call an execution path taken by either a conditional jump or dynamic method dispatching ‘a branch’.

SAC'15 April 13-17, 2015, Salamanca, Spain.
Copyright 2015 ACM 978-1-4503-3196-8/15/04...$15.00.
http://dx.doi.org/10.1145/2695664.2695735

```java
class FileLister {
  File dir;
  List<String> list;
  List<String> args;
  boolean active;
  public static void main(String[] args)  {
      new FileLister().run(args);
  }
  void run(String[] args)  {
      init();
      setArgs(args);
      if (active) addItems();
      printItems();
  }
  void init()  {
      dir = new File(".");
      list = new ArrayList<>();
      args = new ArrayList<>();
      active = false; //defect
  }
  void setArgs(String[] as) {
      for (String a:as) args.add(a);
  }
  void addItems() { // add file names
      List<String> d = this.list;
      for (String n:dir.list())
         d.add(n); // omitted operation
  }
  void printItems() {
      List<String> s = this.list;
      System.out.println(s.toString());
  }
  void addNewItem(String n) { list.add(n); }
 }
```

Figure 2: File lister with an execution omission error.

`false`. This causes an execution omission of the `add` operation, which is expected to be performed. As a result, we observe an execution omission error as an empty value of `y` at the bottom. Such errors are recognized as particularly difficult issues in debugging [20] because the links between an observed error and its defect are missing. Directly chasing back the sources of errors does not lead us to the defect because the error is caused as an indirect effect of a conditional jump.

The proposed omission finder shows the operations that could have been executed if conditional jumps took different branches. However, if we showed all unexecuted code, it would be too large to be useful. A breakthrough is the use of a *points-to analysis* to filter the operations related to the value under inspection and the use of control-flow analysis to relate the filtered operations to actual execution.

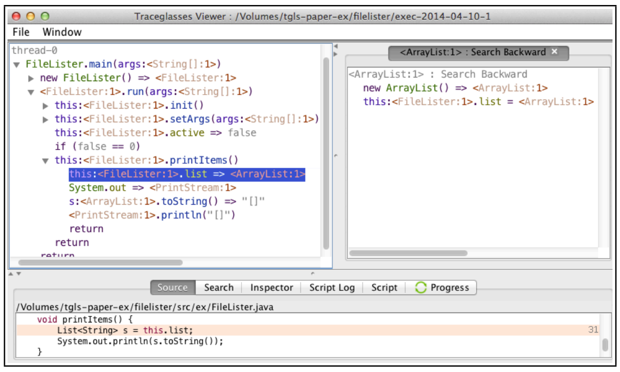
We implemented the omission finder on top of Traceglasses, a fine-grained trace-based debugger for Java [13]. The implementation, which is publicly available,[2] is demonstrated to be useful for debugging execution omission errors in practical open-source programs.

The rest of this paper is organized as follows. Section 2 presents the basic concepts of the trace-based debuggers [13], which serves as a back-end of the omission finder. The section then explains the difficulties in execution omission errors. Section 3 presents the design and implementation of the omission finder. Section 4 evaluates the omission finder with three respects, namely number of inspecting operations, actual debugging times, and execution performances. In Section 5 we discuss related work, and Section 6 discusses several issues facing the omission finder in practice. Section 7 concludes the paper.

2. TRACE-BASED DEBUGGER

2.1 Example Target Program: A File Lister

[2]`http://csl.ec.t.kanazawa-u.ac.jp/~sakurai/project/tgls`


Traceglasses Viewer : /Volumes/tgls-paper-ex/filelister/exec-2014-04-10-1

```
thread-0
▼ FileLister.main(args:<String[]:1>)
  ▶ new FileLister() => <FileLister:1>
  ▼ <FileLister:1>.run(args:<String[]:1>)
    ▶ this:<FileLister:1>.init()
    ▶ this:<FileLister:1>.setArgs(args:<String[]:1>)
      this:<FileLister:1>.active => false
      if (false == 0)
    ▼ this:<FileLister:1>.printItems()
        this:<FileLister:1>.list => <ArrayList:1>
        System.out => <PrintStream:1>
        s:<ArrayList:1>.toString() => "[]"
        <PrintStream:1>.println("[]")
        return
      return
    return
```

```
<ArrayList:1> : Search Backward
  new ArrayList() => <ArrayList:1>
  this:<FileLister:1>.list = <ArrayList:1>
```

```
/Volumes/tgls-paper-ex/filelister/src/ex/FileLister.java
  void printItems() {
      List<String> s = this.list;                                    31
      System.out.println(s.toString());
  }
```

Figure 3: A screenshot of Traceglasses consisting of a tree view (top left), a filtered view (top right), and a source view (bottom).

val ::= [*vname*:] (<*type*:*id*> | "*str*" | *primitive value* | *type*)
event ::= *exec* | *val*.*fld* => *val* | *val*.*fld* = *val*
 | *arith* => *val* | if (*arith*) | return [*val*]
exec ::= *val*.*meth*({*val*,}) [=> *val*] | new *type*({*val*,}) => *val*
arith ::= *uop*(*val*) | *val* *bop* *val*

Figure 4: Syntax of events (abridged). [x] denotes an optional element and $\{x\}$ denotes zero or more repetitions.

Figure 2 shows `FileLister`, an example program that is supposed to list the file names in a current working directory. It actually contains an execution omission error, which we will explain in the next section.

The `main` method of the program creates a `FileLister` instance, and calls the `run` method. The method `run` first sets up instance fields (`init`) and then saves the given arguments (`setArgs`) that are not directly used by the class but rather by a subclass in future. It subsequently gathers file names into the `list` field (`addItems`) and prints them (`printItems`). The `active` field controls whether it invokes `addItems`, which is useful when `FileLister` traverses many directories and gathers file names only from specific directories (not shown in the figure). The class also has an unused method for adding a new item to the list (`addNewItem`), which is used from outside of the class.

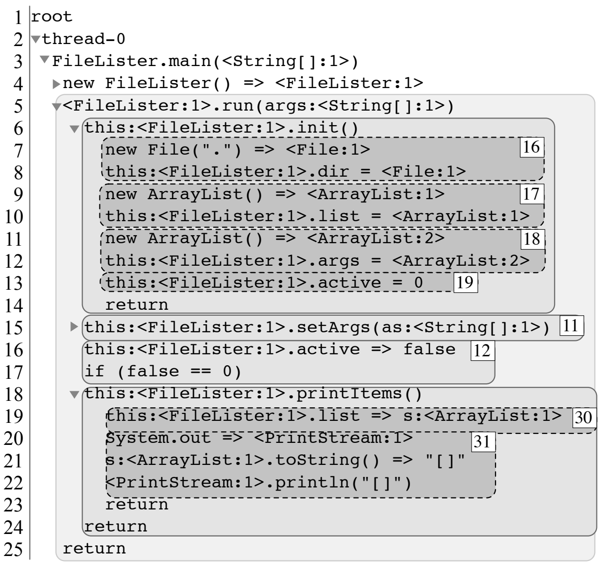
2.2 Trace-Based Debugging with Traceglasses

A trace-based debugger records an *execution history* of a program run and then lets the developer inspect the history afterwards. An execution history is a sequence of *events* (i.e., low-level operations with their parameters) that are performed by the program.

Below, we explain the fundamental concepts of the trace-based debugger by using our Traceglasses debugger [13]. Though it has its own advantages and limitations, the proposed feature can be basically implemented on other trace-based debuggers. The Traceglasses debugger is designed and implemented for Java. It records events at the Java bytecode instruction level and offers three views for inspection, as shown in Figure 3.

2.2.1 Events

An event is a unit of recorded execution and roughly corresponds to the execution of a Java bytecode instruction. Traceglasses display an event in a syntax that resembles the Java source language, as summarized in Figure 4.



```
 1 root
 2 ▼thread-0
 3  ▼FileLister.main(<String[]:1>)
 4   ▶new FileLister() => <FileLister:1>
 5   ▼<FileLister:1>.run(args:<String[]:1>)
 6     ▼this:<FileLister:1>.init()
 7        new File(".") => <File:1>                              16
 8        this:<FileLister:1>.dir = <File:1>
 9        new ArrayList() => <ArrayList:1>                       17
10        this:<FileLister:1>.list = <ArrayList:1>
11        new ArrayList() => <ArrayList:2>                       18
12        this:<FileLister:1>.args = <ArrayList:2>
13        this:<FileLister:1>.active = 0         19
14        return
15     ▶this:<FileLister:1>.setArgs(as:<String[]:1>)  11
16      this:<FileLister:1>.active => false  12
17      if (false == 0)
18     ▼this:<FileLister:1>.printItems()
19        this:<FileLister:1>.list => s:<ArrayList:1>  30
20        System.out => <PrintStream:1>           31
21        s:<ArrayList:1>.toString() => "[]"
22        <PrintStream:1>.println("[]")
23        return
24      return
25    return
```

Figure 5: Events generated from an execution of `FileLister`.

A value (*val*) is a parameter of or a result from an event. It is either an instance (Java object), a primitive value (e.g., numbers and booleans), or a type. An instance is denoted as a pair of its type name and an integer ID unique to each type. Exceptionally, strings are denoted in a literal form surrounded by double quotation marks. When a value is obtained from or assigned to a local variable, the value is denoted with the name of the variable. For example, `v:<FileLister:1>` denotes an instance of `FileLister` accessed through the variable `v`.

There are several kinds of events, ranged over a metavariable *event*, including method or constructor execution (*exec*), field read/write, arithmetic computation (*arith* `=>` *val*), a conditional jump, and return from a method or a constructor. The paper omits other kinds of events, such as array operations and thread synchronizations, for simplicity.

The metavariable *exec* ranges over the events that represent the beginning of a method/constructor execution. Note that an *exec* event denotes the indistinguishable action of calling a method on the caller side and starting the method execution on the callee side. When a method/constructor returns a value, we write the returned value on the right-hand side of `=>`. For example,

```
new FileLister() => v:<FileLister:1>
```

denotes the execution of the constructor of `FileLister`, whose result, `<FileLister:1>`, is assigned to a local variable `v`.

A conditional jump, written as `if (false == 0)`, denotes the values in the conditional expression used for the jump. Since the event is recorded at the bytecode level, evaluation of complex Boolean expression and iteration/exit of a loop are also represented by conditional jumps. Note that a conditional jump event does not contain the branch target because the following events obviously indicate which branch was taken. For the same reason, the execution history does not contain unconditional jumps.

2.2.2 *Tree View*

The tree view (top left in Figure 3) shows all the events in the execution history organized into a tree structure based on the caller-callee relation. Branches of the tree are collapsed by default so as to give an overview of the execution, and can be expanded by clicking.

Figure 5 shows a tree view generated from an execution of `FileLister`. Each rounded box with a number n at its top-right corner corresponds to a statement at line n in Figure 2. The boxes and numbers are added for the explanatory purpose in the paper and are not shown in the actual debugger. Due to space limitation, the bodies of the `FileLister` constructor and the `setArgs` are not shown (denoted as ▶).

As similar to other trace-based debuggers, Traceglasses can list events that are related to a runtime value as in the filtered view (top right in Figure 3). This feature enables developer to trace dataflow easily. Additionally, it can inspect source code (the bottom view in Figure 3) which generated the event selected in the tree view.

2.3 Execution Omission Errors

An execution omission error is an error caused by *not executing* fragments of code that ought to have been executed. This is typically caused by an erroneous value in a conditional jump or a loop condition that surrounds the code fragment, or by an erroneous type in a method invocation.

For example, `FileLister` in Figure 2 causes an execution omission error, which is observed as an empty output. The defect of the error is at line 19, which initializes the `active` field to `false`, instead of `true`. This lets the conditional jump at line 12 skip the branch that adds file names (lines 25–27).

2.4 Difficulties in Debugging Execution Omission Errors

Execution omission errors are difficult to debug even with a trace-based debugger. While trace-based debuggers provide convenient accesses to *executed* operations in the run of a program, they offer no information on operations that are not executed. When we find the field of an instance has an unexpected value, which is the result of an incorrect assignment, a trace-based debugger is useful. However, when the program has missed executing an operation that assigns an expected value to the field, it gives no clue.

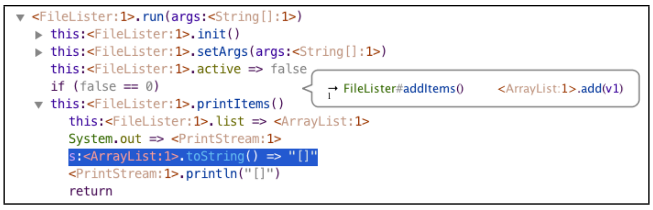
For example, the execution history in Figure 5 indicates that `<ArrayList:1>` is empty[3] at line 21, while it is expected to be a non-empty list of file names as the value of the `list` field obtained at line 19. However, the events that are related to the `ArrayList` before of the line 19 in the execution history are merely the construction events at lines 9–10.

If we knew (1) that the statement `d.add(n)` was not executed due to (2) the conditional branch in `run`, we could have identified the incorrect initialization of the `active` field. However, the fact (1) does not appear in the execution history, and the fact (2) does not directly related to `<ArrayList:1>` in the event history (i.e., the conditional jump at lines 16 and 17).

In practical programs, execution omission errors can have more complicated dependencies hidden in hundreds of thousands of events. These are much harder to reason about.

3. THE OMISSION FINDER

[3]This is because the printed string `"[]"` returned by `toString()`, which describes the contents of the list, is empty.



```
▼ <FileLister:1>.run(args:<String[]:1>)
  ▶ this:<FileLister:1>.init()
  ▶ this:<FileLister:1>.setArgs(args:<String[]:1>)
    this:<FileLister:1>.active => false
    if (false == 0)
  ▼ this:<FileLister:1>.printItems()
      this:<FileLister:1>.list => <ArrayList:1>
      System.out => <PrintStream:1>
      s:<ArrayList:1>.toString() => "[]"
      <PrintStream:1>.println("[]")
      return
```

```
→ FileLister#addItems()      <ArrayList:1>.add(v1)
```

Figure 6: A balloon showing an omitted operation.

We propose the *omission finder* as an additional feature for our trace-based debugger. When the developer has specified an instance in an execution history, it displays operations that could have been performed on the instance if one of the conditional jumps in the history took different branches. When there is any omitted operation that could make the value of the interested instance correct, the developer can move on to reason about the conditional jump that caused the omission.

Next, we show how the omission finder works in Sections 3.1, followed by an overview and detailed description of the algorithm description in Section 3.2 and 3.3. We then discuss the user interface in Section 3.4 and implementation issues in Section 3.5.

3.1 Debugging with the Omission Finder

The following steps illustrate how the omission finder works.

(1) The developer uses a trace-based debugger to find the defect of an observed error. He or she finds that (the field of) an instance has an unexpected value. In Figure 5 an observed error is empty `<ArrayList:1>` at line 22.

(2) The developer isolates the events to the ones related to the instance, expecting to find events that created the unexpected value. When the value of the instance was originally correct, yet eventually incorrect, we doubt an execution omission error. In Figure 5, filtering the events with respect `<ArrayList:1>` merely yields the construction of the array. Though it was correct at the time of construction, it was not correct at the time of printing. Hence, we suspect an execution omission error with respect to that array.

(3) The developer runs the omission finder on the instance in question. The finder then displays balloons on several events in the history as shown in Figure 6. Each balloon shows a list of operations that were not executed, but that could have operated on the specified instance if executed. A balloon is attached to the conditional jump that prevented the execution of the shown operations. We call such jump a *dominant conditional jump*. The balloon in Figure 6 tells us that the `add` method could have been invoked on `<ArrayList:1>` in the `addItems` method if the conditional jump took the other branch. In the source program (Figure 2), we can find such a method call at line 27.

(4) Among the operations shown in the balloons, the developer identifies an operation that could have made the instance have a correct value.

(5) The developer then investigates why the identified operation was not executed by looking into the conditional jump at which the balloon is attached. In Figure 6, the balloon is attached to a conditional jump in `addItems`. The previous event shows the condition was obtained from the `active` field. From these, we can conclude that the field had been incorrectly initialized.

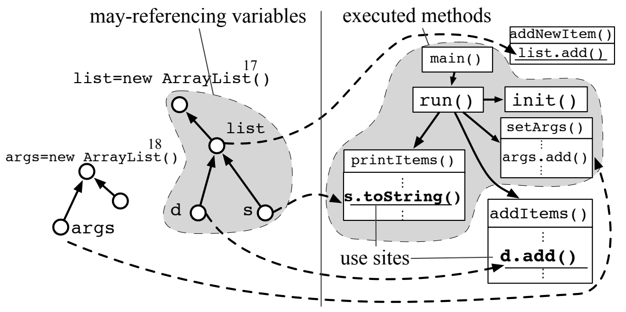


Figure 7: Identifying use sites by using a pointer assignment graph (PAG, left) and a call graph (CG, right).

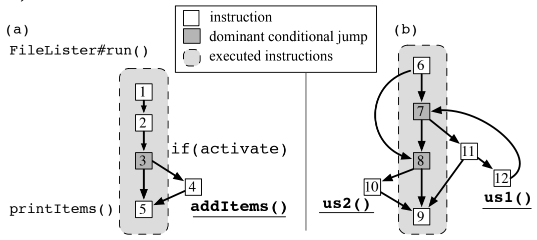


Figure 8: Dominant conditional jumps in control-flow graphs (CFGs) constructed from (a) `run()` in FileLister and (b) a more complicated method with conditionals and a loop.

3.2 Overview of the Omission Finder Algorithm

The omission finder's algorithm is to find an accurate set of omitted operations, and to display those operations at the appropriate positions. To do this, we use a *pointer assignment graph* (PAG), a *call graph* (CG), and *control flow graphs* (CFGs). Below, we explain the difficulty of finding an accurate set of omitted operations, and then present an overview of the algorithm by using Figures 7 and 8.

Even though we can easily think of several naive approaches to filter out omitted operations, those approaches would list too many operations to be used for debugging. For example, if it just showed all unexecuted operations, they would be undoubtedly too many. Even if it shows only unexecuted operations related to the type of the interested value (i.e. if we are interested in an `Vector` instance, list all the operations that takes an array as a target of a parameter), there would be many unrelated operations for generic types like `Vector`.

To cope with the difficulty, we combine static *dataflow information* with a dynamic execution history. A points-to analysis can statically tell us where an instance will flow to (i.e., the dataflow). By using the results of points-to analysis, we can dramatically reduce the number of candidate operations. In addition, by exploiting the static inter- and intra-procedural control flow information, we can relate those operations with events in a dynamic execution history.

Given an instance in an execution history, the algorithm first finds *use sites*, which are the instructions in the program that could have used the instance, by performing the following three operations. (1) In the execution history, it looks for the `new` instruction that constructed the instance. (2) By using a PAG (e.g., left-hand side of Figure 7, it finds local variables that may reference instances constructed by the `new` instruction (the nodes in the shaded area). (3) By using a CG (e.g., the right-hand side of Figure 7), it finds instructions that use one of those local variables; i.e., the instructions that could have operated on the given instance (`s.toString()` in `printItems()` and `d.add()` in `addItems()` in the figure).

The algorithm then constructs *call chains*, each of which is a sequence of method calls from an executed method to a method that contains one of the use sites. A simple graph search can construct the call chains. From the executed methods (those in the shaded area) on the right-hand side of Figure 7, we obtain five call chains, namely, $\langle \texttt{main}, \texttt{run}, \texttt{printItems}\rangle$, $\langle \texttt{run}, \texttt{printItems}\rangle$, $\langle \texttt{printItems}\rangle$, $\langle \texttt{main}, \texttt{run}, \texttt{addItems}\rangle$ and $\langle \texttt{run}, \texttt{addItems}\rangle$.

The algorithm finally identifies, for each executed method, *dominant conditional jumps*, which are the executed conditional jumps that prevented the execution of a use site. Such jumps can be characterized by using a CFG of the method, as illustrated in Figure 8.

In the CFG, a dominant conditional jump is an executed branching node that has an unexecuted successor path to a use site, or to an invocation of the next method in a call chain that starts from the current method. In the CFG of the `run()` method (Figure 8 (a)), the conditional jump node $\boxed{3}$ is the dominant conditional jump. In Figure 8 (b), the nodes $\boxed{7}$ and $\boxed{8}$ are the dominant conditional jumps among four conditional jumps, given `us1()` and `us2()` are the methods containing the use sites.

Having computed the dominant conditional jumps, the omission finder displays "`d.add(n)` in `addItems`" next to the dominant conditional jump.

3.3 The Omission Finder Algorithm

We present the omission finder algorithm, which consists of the functions of computing use sites, call chains, and dominant conditional jumps, as well as the main algorithm.

3.3.1 Definition

Before presenting the main algorithm, we first define the properties of the source program and the execution history, and the auxiliary functions.

The algorithm assumes a program is a set of methods (m) given in a bytecode format. The metavariable i ranges over instructions, each of which is distinguished by an enclosing method, denoted as $\mathit{Encl}(i)$, and an address in the method. $\mathit{Instructions}(m)$ is a set of all instructions in the method m.

A *pointer assignment graph* (PAG) represents interprocedural dataflow information in a program. A node in PAG is either a variable (v), an instance ($\texttt{new}\ \mathit{type}^a$), or an instance field ($v.\mathit{fld}$). Instance nodes are distinguished by allocation sites (a) in the source code. In other words, two instances constructed by the same `new` expression in the source code are considered as the same instance node in PAG.

A PAG edge represents a relation such that the destination is assigned to the origin. Therefore, when there is a path from a variable to an instance in the PAG, the variable can have a reference to the instance.

A *call graph* (CG) is a directed graph among method definitions in the program. An edge in CG denotes the existence of a method call in the originating method definition to the destination method.

A control-flow graph of method m, denoted as $\mathit{CFG}(m)$, is a directed graph among machine instructions in m. An edge denotes that the destination instruction can be executed after the originating instruction. We denote a set of all acyclic paths in graph G as $\mathit{Paths}(G)$.

Events are the set of events recorded during an execution of the program. The metavariable *event* ranges over *Events*. $\mathit{InstructionsUsing}(v)$ is a set of instructions in the entire program that use a local variable v as a parameter. $\mathit{Source}(\mathit{event})$ gives the instruction that generated *event*. *Execs* is a subset of *Events* that contains all events that execute either methods or constructors (as denoted by *exec* in Figure 4) and no other kinds of events. $\mathit{Method}(\mathit{exec})$ gives the name of the method executed by *exec*. $\mathit{Body}(\mathit{exec})$ is a set of events that are directly executed in the body of *exec*.

3.3.2 The Algorithm

We first define several auxiliary functions. First, given an instance ID *oid*, we obtain a set of a pair of instruction i and local variable v such that i may operate on the instance through v. On the PAG, the instance is represented by an allocation site. We therefore obtain the following function:

$$\begin{aligned}
&\mathit{UseSites}(\mathit{oid}) = \\
&\quad \{(i,v) \mid \forall\langle v,\ldots,l\rangle \in \mathit{Paths}(\mathit{PAG}).\ l = \texttt{new}\ \mathit{type}^a, \\
&\qquad\qquad \forall i \in \mathit{InstructionsUsing}(v)\} \\
&\quad \text{where } \texttt{new}\ \mathit{type}^a\texttt{(...)}\ \texttt{=>}\ \mathit{oid} \in \mathit{Events}.
\end{aligned}$$

For example, when *oid* is `<ArrayList:1>` in Figure 7, we obtain (`s.toString()`, `s`), (`d.add()`, `d`) and (`list.add()`, `list`) as the use sites.

Given a method m_0 and use sites us, we find call chains (i.e., a sequence of methods) from m_0 to a method that contains an instruction in us. When m_0 contains an instruction in us, a chain containing merely m_0 is also included in the result. We limit the result to call chains shorter than a threshold, as our previous experience suggests that relevant operations often exist at a few method-call distance. The threshold is also useful to control the trade-off between precision and time overhead (The current value is 5).

$$\begin{aligned}
&\mathit{CallChain}(m_0, \mathit{us}) = \\
&\quad \{(\langle m_0,\ldots,m_n\rangle, i, v) \mid \forall\langle m_0,\ldots,m_n\rangle \in \mathit{Paths}(\mathit{CG}). \\
&\qquad\qquad n \leq \mathit{THRESHOLD}, \\
&\qquad\qquad \forall(i,v) \in \mathit{us}.\ m_n = \mathit{Encl}(i)\} \\
&\quad \cup\ \{(\langle m_0\rangle, i, v) \mid \forall(i,v) \in \mathit{us}.\ m_0 = \mathit{Encl}(i)\}
\end{aligned}$$

For example, the call chains from `run()` to the use sites (the underlined instructions) in Figure 7 are
$(\langle \texttt{run}, \texttt{printItems}\rangle, \texttt{s.toString()}, \texttt{s})$ and
$(\langle \texttt{run}, \texttt{addItems}\rangle, \texttt{d.add()}, \texttt{d})$.

Given an execution event *exec*, the next function determines a set of instructions that are not executed in *exec*'s body.

$$\begin{aligned}
\mathit{Unexecuted}(\mathit{exec}) = {} & \mathit{Instructions}(\mathit{Method}(\mathit{exec})) \\
& \setminus \textstyle\bigcup_{\forall \mathit{event} \in \mathit{Body}(\mathit{exec})} \mathit{Source}(\mathit{event})
\end{aligned}$$

A dominant conditional jump in a method execution event (*exec*) with respect to an omitted instruction (i_o) in the method is a conditional jump event in the history whose untaken branch can lead to i_o only by following unexecuted instructions.

$$\begin{aligned}
&\mathit{DominantJumps}(\mathit{exec}, i_o) = \\
&\quad \{e_s \mid \forall e_s \in \mathit{Body}(\mathit{exec}). \\
&\qquad\quad \forall\langle i_1,\ldots,i_k\rangle \in \mathit{Paths}(\mathit{CFG}(\mathit{Method}(\mathit{exec}))). \\
&\qquad\quad \mathit{Source}(e_s) = i_1,\ i_k = i_o, \\
&\qquad\quad \forall j \in \{2,\ldots,k\}.\ i_j \in \mathit{Unexecuted}(\mathit{exec})\}
\end{aligned}$$

```
   input $oid$: an instance ID
1  for $\forall exec \in Execs$ do
2   for $\forall(\langle m_0, \ldots, m_n\rangle, i, v)$
3          $\in CallChain(Method(exec), UseSites(oid))$ do
4    if $n \geq 1$ then
5      $I_o = \{i_t \mid \forall i_t \in Instructions(m_0).\ i_t = \texttt{invoke}\ m_1\}$
6    else
7      $I_o = \{i\}$
8    for $\forall i_o \in I_o$ do
9      for $\forall e_s \in DominantJumps(exec, i_o)$ do
10       display "$i[oid/v]$ is in $m_n$ after $n$ calls"
```

Figure 9: Main algorithm of the omission finder.

In other words, if the condition was reversed, execution of such a jump could have resulted in the execution of i_o. Such an event can be found in the body events of *exec* that have a control flow to i_o whose instructions in the flow were not executed during the method execution. For example in Figure 8 (b), the dominant conditional jumps with respect to the execution of `invoke us1` ($\boxed{\boxed{12}}$) and `invoke us2` ($\boxed{\boxed{10}}$) are the events that executed instructions $\boxed{7}$ and $\boxed{8}$, respectively.

The main algorithm, which is shown in Figure 9, takes an instance ID (*oid*) in the event history as an input.

The algorithm first iterates over all method executions (line 1) and constructs call chains from each executed method to the use sites of *oid* (lines 2 and 3).

Next, for each call chain, it selects omitted instructions I_o in the current method m. If the call chain has more than one method, I_o are method invocations of the next method in the chain (lines 4 and 5). Otherwise, I_o is the use site itself (line 7). For example, a call chain $(\langle\texttt{run}, \texttt{addItems}\rangle, \texttt{d.add()}, \texttt{d})$ selects `invoke addItems` in `run()` as an omitted instruction (i_o), which leads the execution to the use site `d.add()`.

The rest of the algorithm identifies the dominant conditional jumps (line 10) for each omitted instruction and displays a balloon along with the jump (line 11). When displaying an omitted instruction (e.g., `d.add()`), it replaces the variable with the instance so that the developer can see how the instance is used as which parameter in the instruction (e.g., `<ArrayList:1>.add()`).

3.4 User Interface

The user interface component, which displays the balloons computed by the above algorithm, reduces the number of balloons by (1) filtering obvious operations out and (2) summarizing duplicated operations.

The filtering feature hides balloons that contain method call operations to specific methods. When a program manipulates an instance, there are usually both read and write operations. As only write operations can only be meaningful as omitted operations, filtering read operations will effectively reduce the amount of information. Our current implementation merely lets the developer select uninteresting operations (such as `ArrayList.size()`) and then all balloons that contain the same method calls will be hidden.

The summarizing feature bundles redundant balloons into one. When an omitted operation is reachable from different dominant conditional jumps, a balloon is attached to each jump. When those conditional jumps are folded in a tree view, those balloons will be bundled into one and attached to a method call event.

3.5 Implementation

We realized the omission finder by extending Traceglasses, which we implemented in our previous work [13]. The entire system consists of approximately 60,000 lines of Java code and is publicly available[4]. The extension for the omission finder amounts to approximately 11,600 lines.

The Traceglasses debugger collects an execution history by instrumenting a Java bytecode program. The instrumented program dumps, when it runs, information of executed instructions, or events, into several files, namely (1) an event database consisting of a sequence of executed bytecode instructions with parameters, (2) an index file containing caller and callee relations, and (3) an index file that maps an instance ID to a set of events.

Our current implementation uses the Soot framework [16] for obtaining static program analysis information. As for constructing a pointer assignment graph (PAG), we used the default algorithm in the Soot Pointer Analysis Research Kit (Spark) [7]. However, our algorithm does not depend on a specific implementation of points-to analysis—we can easily switch the analysis to other advanced ones, such as Paddle [8], DOOP [2], and others [6, 14, 19], when we need more efficiency, more precision, or an incremental nature.

4. EVALUATION

We evaluated the omission finder with respect to (1) the feasibility when it is applied to a real-world program with an execution omission error, (2) the actual times of debugging processes through a user study, and the performance of the instrumented programs as well as the omission finder algorithm.

4.1 Debugging Apache Ant with the Omission Finder

In order to evaluate the effectiveness of the omission finder, we analyzed a debugging process of a real-word program, Apache Ant[5], with a known execution omission error (bug #40722). The operations suggested by the omission finder were positive, i.e., including the actual defect, and much fewer (13) than the number of operations that would need to be inspected without the omission finder (134 conditionals in 655 events). The authors conclude that the omission finder can be useful to reduce the number of debugging steps when there is an execution omission error.

4.2 Comparison of Actual Debugging Times

We carried out a controlled experiment to compare debugging times by users with and without the omission finder. We have the following three research questions.

RQ1: When there is an execution omission error, does the omission finder shorten the debugging time?

RQ2: When there is no execution omission error, does the omission finder have a negative impact on a debugging time?

RQ3: Are debugging times with trace-based debuggers and those with break-point debuggers comparable? (This question is about validity of trace-based debuggers, rather than the omission finder.)

We prepared four debugging tasks, and measured participants' task completion times by using the following three de-

[4]`http://csl.ec.t.kanazawa-u.ac.jp/~sakurai/project/tgls`

[5]`http://ant.apache.org`.

buggers: (OF) Traceglasses with the omission finder, (TG) Traceglasses without the omission finder, and (BD) a break-point debugger provided in Eclipse 4.2.

4.2.1 *Debugging Tasks*

Each of the debugging tasks is to, given a program that produces an erroneous output, find out a source code location of the defect. We constructed the defective programs by manually modifying open source programs in order to control the size of the programs suitable to the user study.

We prepared 4 defective programs from three open source programs namely SCXML [6], NekoHTML[7], and Apache Ant. We refer those programs as TC1, 2, 3 and 4. The lengths of execution traces from TC1–4 are 217, 9422, 155535, and 5596, respectively. TC4 contains an error that is *not* caused by execution omission, which is prepared for answering RQ2.

The debugging times in the following experiments do not include the times for collecting events and the times for constructing PAGs. We prepared those data before conducting experiments.

4.2.2 *Method*

24 computer science major students at two universities (10 undergraduates and 14 graduates) participated in the experiment. They have from 2 to 8, 3.3 on average, years experience of Java programming. None of them have ever seen the programs to be debugged. 20 of them have an experience of using a break-point debugger.

We made three groups of participants. For each group, we assigned a different debugger to each task so that each task-debugger pair has a roughly equal number of participants.

Before working on the tasks, the participants are requested to follow a tutorial session with the tree debuggers. We used a simple program with an execution omission error similar to `FileLister` in the paper.

Each participant then worked on each of the tasks one by one with the assigned debugger. When a participant could not find the defect in a task within one hour, he or she was requested to proceed to the next task.

4.2.3 *Results*

For each pair of a TC and a debugger, we counted the number of successful/failed participants as well as average debugging times. Then, for each TC, we tested whether debugging times with two debuggers are statistically different based on the Mann-Whitney U-test.

OF vs. TG: When execution traces are long (TC2 and 3), OF gives higher success ratios than TG (7/7 and 8/8 with OF vs. 5/7 and 3/6 with TG), and shorter debugging times (461 and 1139 secs. with OF vs. 1418 and 3513 secs. with TG). The differences are statistically significant with 97% confidence. When the execution trace is short (TC1), success ratios and debugging times with OF and TG are not significantly different. Without an execution omission error (TC4), the debugging with OF (1974 sec.) was longer than that with TG (1067 sec.), but the difference is not statistically significant even with 95% confidence.

TG vs. BD: For all the tasks, TG gives the same or slightly higher success ratios (8/8, 5/7, 3/6 and 5/5) than BD does (5/5, 3/5, 3/7 and 5/5), and shorter or slightly longer debugging times (888, 1418, 3513 and 1067 secs. vs. 1420, 2408, 3600 and 924 secs.), though the differences are not statistically significant except for TC1.

[6]`http://commons.apache.org/scxml/`

[7]`http://nekohtml.sourceforge.net`

With those results, we answer our research questions:

RQ1: When there is an execution omission error, the omission finder can shorten the debugging time when the length of an execution history is not short.

RQ2: When there is no execution omission error, the omission finder could give a negative impact on debugging time, but not validated by our experiments.

RQ3: Debugging times with the trace-based debuggers and with the break-point debugger are comparable.

4.3 Performance Measurement

Though the performance is not the paper's primary concern, we measured (1) overheads of instrumented programs, and (2) execution times of the omission finder algorithm. As for (1), execution times of 5 real-world Java programs instrumented by Traceglasses were slower than the original programs by the factors of 19 to 62.5, whereas those with Whyline [4] were from 5 to 25 but one failed due to out of memory. As for (2), execution of the omission finder for SCXML (Section 4.2.1) took 139.4 seconds and 1531 MB heap memory. We conclude that the overheads of Traceglasses and the omission finder are acceptable to be applied to real-world programs.

5. RELATED WORK

Zhang [20] et al. proposes a mutation-based technique to debug an execution omission error. It repeatedly generates a mutant of the program by inverting one of the conditional jumps until it finds a mutant that passes a test case. While it is an automated approach, it will only succeed when a correct result can be obtained by merely inverting conditional jumps. It also requires an enormous number of executions.

As far as the authors know, existing trace-based (omniscient or back-in-time) debuggers, including ODB [5], TOD [11], STIQ [12], Unstack [3], Traceglasses [13] and Compass [10, 9], do not support execution omission errors.

Whyline [4] is a trace-based debugger that can answer questions why a variable did not have an expected value. However, as far as the authors tested, Whyline works only when an expected value is overwritten by another value. In other words, when a variable has an unexpected value due to an execution omission error, Whyline cannot provide appropriate answers.

Several dynamic program slicing techniques [18] take omitted execution into account, though they have been developed for different purposes than debugging.

Agrawal et al. proposed the relevant slicing technique for narrowing regression test cases [1]. When a program is changed, it checks if the change affects each test case by checking overlap between a dynamic program slice of the previous program and the changed code fragments. Gyimóthy et al. improved the accuracy by including unexecuted branches of change-affected conditional jumps in the slice [15]. In other words, they incorporated omitted executions into the control dependency in order to improve the accuracy. Wang and Roychoudhury proposed an efficient dynamic slicing method based on an execution trace at the Java bytecode level [17]. Their method extends the relevant slicing algorithm with a points-to analysis.

6. DISCUSSION

Severity of Execution Omission Errors.

One might argue that execution omission errors are infrequent. Indeed, our investigation into several open source programs identified relatively few execution omission errors.

However, we believe it is important to deal with bugs that could take a long time to debug, even if they are infrequent. As we showed in Section 4.2, debugging programs with execution omission errors take much longer times with existing debuggers in practice.

Moreover, we should be careful with the types of bugs that we can find in repositories of open source programs. Since the developer usually commits a program to a repository after running test cases, committed programs rarely have errors that are detectable by test cases. We suspect that execution omission errors are found more frequently during development of a program.

When to Use the Omission Finder.

Since the omission finder has some overhead, it should be used only when the error is caused by execution omission. As we explained in Section 3.1, a developer should use the omission finder only when (1) an instance has an unexpected value, and (2) the value was initially correct; i.e., we expect that the instance should have been modified to have a correct value. Since there is no approach to such a case other than examining every event, the overhead of the omission finder would be acceptable.

Limitation.

Our algorithm does not detect omitted operations if a loop lacking with an iteration caused an error. This is because the algorithm searches omitted operations in unexecuted instructions. We however believe that this limitation is not a serious problem in most cases because the developer can find the operations executed during the last iteration.

The omission finder suggests only code fragments that are *totally omitted*, which have never been executed in a run. Therefore, partially omitted code fragments, which are executed for less number of times than expected, are not suggested. This kind of partial omission could happen, for example, when a boundary condition of a loop is not correct. However, partial omission is less serious than total omission. When we found a collection that should have ten but nine elements, we can easily find the code that added the nine elements and then we would be able to reach the loop condition around the code.

7. CONCLUSION

We proposed the omission finder that suggests operations that could have been performed on an instance if a program had behaved differently. The key idea is to correlate an execution history with static analysis—namely, a pointer assignment graph, a call graph, and control-flow graphs—to identify such operations and relevant conditional jumps.

We implemented the omission finder on top of Traceglasses, a trace-based debugger for Java. An application of the omission finder to a real-world program with an execution omission bug demonstrated that it correctly suggests omitted operations over a relevant condition branch, hence it can reduce the number of debugging steps in practice. Our case study with 24 participants with three different debuggers also showed that the omission finder significantly reduces the times of debugging when there are execution omission errors in long execution histories.

8. ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Numbers 23700041, 26330078 and 26730034.

9. REFERENCES

[1] H. Agrawal et al. Incremental regression testing. *ICSM '93*, pp. 348–357, 1993.

[2] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. *OOPSLA '09*, pp. 243–262, 2009.

[3] C. Hofer et al. Design and implementation of a backward-in-time debugger. *NODe/GSEM '06*, pp. 17–32, 2006.

[4] A. Ko and B. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. *ICSE '08*, pp. 301–310, 2008.

[5] B. Lewis. Debugging backwards in time. *AADEBUG '03*, 2003.

[6] O. Lhoták and K.-C. A. Chung. Points-to analysis with efficient strong updates. *POPL '11*, pp. 3–16, 2011.

[7] O. Lhoták and L. Hendren. Scaling Java points-to analysis using SPARK. *CC '03*, pp. 153–169, 2003.

[8] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *TOSEM*, 18:3:1–3:53, 2008.

[9] A. Lienhard et al. Practical object-oriented back-in-time debugging. *ECOOP '08*, pp. 592–615, 2008.

[10] A. Lienhard et al. Flow-centric, back-in-time debugging. *TOOLS '09*, pp. 272–288, 2009.

[11] G. Pothier et al. Scalable omniscient debugging. *OOSPLA '07*, pp. 535–552, 2007.

[12] G. Pothier and É. Tanter. Summarized trace indexing and querying for scalable back-in-time debugging. *ECOOP '11*, pp. 558–582. 2011.

[13] K. Sakurai et al. Traceglasses: A trace-based debugger for realizing efficient navigation. *IPSJ Transaction on Programming*, 3(3):1–17, 2010.

[14] M. Sridharan et al. Demand-driven points-to analysis for Java. *OOPSLA '05*, pp. 59–76, 2005.

[15] G. Tibor et al. An efficient relevant slicing method for debugging. *ESEC/FSE-7*, pp. 303–321, 1999.

[16] R. Vallée-Rai et al. Soot - a Java optimization framework. *CASCON '99*, pp. 125–135, 1999.

[17] T. Wang and A. Roychoudhury. Dynamic slicing on Java bytecode traces. *TOPLAS*, 30:10:1–10:49, 2008.

[18] M. D. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[19] G. Xu et al. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. *ECOOP '09*, pp. 98–122, 2009.

[20] X. Zhang et al. Towards locating execution omission errors. *PLDI '07*, pp. 415–424, 2007.