Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Original software publication

NOD4J: Near-omniscient debugging tool for Java using size-limited execution trace

Kazumasa Shimari [a,*], Takashi Ishio [b], Tetsuya Kanda [a], Naoto Ishida [a], Katsuro Inoue [a]

[a] *Graduate School of Information Science and Technology, Osaka University, Osaka, Japan*
[b] *Graduate School of Science and Technology, Nara Institute of Science and Technology, Nara, Japan*

A R T I C L E   I N F O

Article history:
Received 15 April 2020
Received in revised form 17 February 2021
Accepted 18 February 2021
Available online 25 February 2021

Keywords:
Dynamic analysis
Logging
Software visualization

A B S T R A C T

Logging is an important feature of a software system to record run-time information. Detailed logging allows developers to collect run-time information in situations where they cannot use an interactive debugger, such as continuous integration and web application server cases. However, extensive logging leads to larger execution traces because few instructions can be repeated many times. This paper presents our tool NOD4J, which monitors a Java program's execution within limited storage space constraints and annotates the source code with observed values in an HTML format. Developers can easily investigate the execution and share the report on a web server. We show two examples that our tool can debug defects using incomplete execution traces.

* Corresponding author.
E-mail addresses: k-simari@ist.osaka-u.ac.jp (K. Shimari), ishio@is.naist.jp (T. Ishio), t-kanda@ist.osaka-u.ac.jp (T. Kanda), n-isida@ist.osaka-u.ac.jp (N. Ishida), inoue@ist.osaka-u.ac.jp (K. Inoue).

https://doi.org/10.1016/j.scico.2021.102630

Software metadata

Software metadata description	
Current software version	v0.2.3
Permanent link to executables of this version	Recorder: https://github.com/takashi-ishio/selogger/releases/tag/v0.2.3 Post-processor and Interactive View: https://github.com/k-shimari/nod4j/releases/tag/v0.2.3
Legal Software License	The MIT License (MIT)
Computing platform/Operating System	Recorder: Any operating system where Java runtime is available Post-processor and Interactive View: Any operating system where Java and Node.js are available
Installation requirements & dependencies	Java SE Runtime Environment, npm, Node.js
If available Link to user manual	https://github.com/takashi-ishio/selogger/blob/develop/README.md & https://github.com/k-shimari/nod4j/blob/master/README.md
Support email for questions	k-simari@ist.osaka-u.ac.jp

Code metadata

Code metadata description	
Current code version	v0.2.3
Permanent link to code/repository used of this code version	https://github.com/ScienceofComputerProgramming/SCICO-D-20-00066
Legal Code License	The MIT License (MIT)
Code versioning system used	git
Software code languages, tools, and services used	Java, TypeScript, React, ANTLR
Compilation requirements, operating environments & dependencies	Java SE Development Kit, Apache Maven
If available Link to developer documentation/manual	(N/A)
Support email for questions	k-simari@ist.osaka-u.ac.jp

1. Introduction

Debugging is a methodology used to identify defects in the source code by diagnosing its external behavior. For efficient debugging, developers monitor the execution order of instructions and actual values of variables in the source code [1]. Additionally, developers may compare program behavior at the various points of execution where a failure does or does not occur [2,3]. Interactive visualization tools such as JIVE [4] and break-point debuggers are useful for such analysis; however, developers struggle to use these tools for systems that run on continuous integration and web application servers.

Logging is a common practice that is used to record a program execution as a sequence of messages that report a software's progress and its important data [5]. However, logs recorded in production may not contain sufficient data for debugging because the data to be logged is determined at development time [6]. Therefore, to enable efficient debugging, an automatic method capable of recording a detailed program's execution is needed.

Omniscient debugging [7] is a method that can be used to record all the runtime events during program execution. Although the method enables developers to inspect the state of a program at an arbitrary point in execution, it results in a huge execution trace, and in some cases, grows as fast as 10 MB per second [8]. Developers have difficulty estimating the size of an execution trace prior to execution. Therefore, this implies difficulty in determining what data should be logged to fix bugs in a deployed environment [9].

In our previous work, we proposed Near-Omniscient Debugging for Java using size-limited execution traces [10]. Since a full execution trace includes many uninteresting method calls such as utility functions [9], we introduce a parameter k that specifies the maximum number of recorded values for each instruction. This parameter limits the size of an execution trace for repeatedly executed instructions, while keeping all actual values of the variables associated with instructions that are executed less than k times. Our result shows that 1% of complete execution traces enabled us to obtain data dependencies with a precision of 91.8% and recall of 79.0% from the traces.

In this work, we present a tool NOD4J (Near-Omniscient Debugger for Java), which records and visualizes an execution trace within limited storage space constraints. The tool records local variables and fields used in a Java program execution and annotates the source code with the recorded values. We describe the implementation details and show two usage examples of debugging actual bugs.

In the remainder of the paper, Section 2 explains the background of the tool and Section 3 describes its implementation. Then, Section 4 describes usage examples. Finally, Section 5 concludes the paper and describes future work.

2. Background

Inspecting the runtime behavior of a program is an important activity for debugging. To achieve this goal, omniscient debugging records all the runtime events such as memory access inside a program. Similarly, Record-and-Replay methods [11–13] record all the interactions between a program and its external environment. As with omniscient debugging, the approach may result in a huge execution trace [13].

One of the causes for larger execution traces is the repetition of the program instructions. To reduce the size of the execution traces, various compression and sampling methods have been proposed. Wang et al. [14] proposed an effective compression method tailored for execution traces comprising a sequence of memory addresses accessed by a program. Cornelissen et al. [9] reported that execution traces excluding unimportant utility functions retain more information than a trace filtered by a simple sampling algorithm using the same storage space. Hizrel et al. [15] proposed Bursty Tracing, a sampling method that periodically turns monitoring on and off. However, users of those methods cannot estimate the size of a trace prior to a program execution. NOD4J enables users to specify the size of an execution trace.

NOD4J is a tool to visualize the values of variables recorded in a program execution. Exiting tools, JIVE and Querypoint, provide similar features. JIVE is an interactive execution environment for Eclipse that visualizes a Java program execution at runtime [4]. While JIVE adds useful features to Eclipse debugger, it cannot visualize a program execution outside of the debugger such as continuous integration. Querypoint is a Firefox plugin, which provides critical information for debugging JavaScript programs [16]. This tool provides a program location where a questionable value was assigned. Instead of recording an execution trace, Querypoint executes the program twice; The first execution observes values and the second execution identifies the point where a questionable value was assigned.

3. NOD4J overview

Our tool named NOD4J records a partial execution trace of a Java program and generates an HTML-based view to interactively explore the recorded trace on a web browser. Fig. 1 illustrates the key idea of our tool. In this figure, we have a limited storage space that can record up to six steps of program execution. Suppose an execution comprises nine steps, as indicated by numbers in yellow boxes. A naive time-series logging records the last six steps as indicated in the bottom of the figure. On the other hand, our tool prepares buffers for each line of code in order to record the latest step for each line, as shown on the right side of the figure. This approach discards an execution trace for repeated instructions in the loop but retains the other information completely. By recording the latest observed values, abnormal behaviors are likely recorded in case a program crashes. Because the trace retains the program's initialization process which is executed only once, developers can also analyze the configuration parameters of the execution.

Our tool comprises three components: trace recorder, post-processor, and interactive view. The trace recorder component records an execution trace of a Java program in storage. The post-processor component links the recorded trace to the source files of the program. The interactive view shows the source code contents annotated with trace information. The following subsections explain each component in detail.

3.1. Trace recorder

Our recorder component named SELogger (Logger for Software Engineering research) is an extension of the existing trace recorder for REMViewer [17], an omniscient debugging tool. It is implemented as a Java agent working inside a Java Virtual Machine. During program execution, the recorder monitors class loading events and injects logging instructions into the loaded classes using ASM, a Java bytecode manipulation framework.[1] The injected logging instructions are executed as a part of the target program. The standard library classes (e.g., `java` and `javax` packages) are excluded from injection in order to avoid license issues.

Table 1 shows a list of major runtime events recorded by the component. To capture inter-procedural control-flow in a program execution, the recorder component uses two types of events defined in AspectJ [18]: Method Execution and Method Call. A Method Execution event is recorded when a method body is executed (i.e., on a callee side), while a Method Call event is recorded before a method invocation instruction is executed (i.e., on a caller side). Fig. 2 shows an example code fragment indicating the locations of logging instructions for `methodE`. In the code fragment, a Call event is recorded at line 3, and then `methodE` is called at line 4. The Call event is followed by a Method Entry event of an actually executed method depending on the type of `this` object; line 10 records a Method Entry event if the method is selected by dynamic binding. When the execution of the method is finished, a Method Normal Exit event representing the location of a return statement is recorded. Finally, a Call Return event is recorded on the caller side. The sequence of events enables us to trace the actual execution path determined at runtime. In addition to the control-flow information, their arguments, return values, and exceptions are also recorded in an execution trace. It is worth noting that a single instruction may be recorded as multiple runtime events; for example, a method execution is represented by a Method Entry event recording a receiver object and a number of Method Param events recording the parameters.

The recorder component also records three types of memory access: local variables, fields, and arrays. The Local Load and Local Store events record actual values read from and written to local variables in a trace. The Local Increment event represents a pair of Local Load and Local Store performed by an increment instruction (e.g. "`i++`") that frequently appears in a program. Events in the Field Access category record object references in addition to values read from and written to

[1] https://asm.ow2.io/.

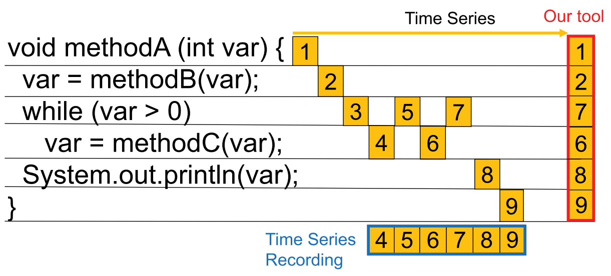


Fig. 1. An execution trace of NOD4J recording k events for each instruction ($k = 1$). (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

Table 1
Major runtime events recorded by the recorder component.

Event Category	Event Name	Recorded data
Method Execution	Method Entry	Receiver object
	Method Param	Parameter given to the method
	Method Normal Exit	Returned value from the method
	Method Exceptional Exit	Exception thrown by the method
Method Call	Call	Receiver object
	Call Param	Parameter passed to the callee
	Call Return	Returned value from the callee
Local Variable	Local Load	Value read from the variable
	Local Store	Value written to the variable
	Local Increment	Value written to the variable by an increment instruction
Field Access	Get Instance Field	Object whose field is read
	Get Instance Field Result	Value read from the field
	Put Instance Field	Object whose field is written
	Put Instance Field Value	Value written to the field
Array Access	Array Load	Accessed array to read
	Array Load Index	Accessed index to read
	Array Load Result	Value read from the array
	Array Store	Accessed array to write
	Array Store Index	Accessed index to write
	Array Store Value	Value written to the array
	Array Length	Array whose length is referred
	Array Length Result	The length of the array

```
void methodD() {
    ...
    // record a Call event of methodE
    this.methodE();
    // record a Call Return event of methodE
    ...
}

void methodE() {
    // Record a Method Entry event of methodE
   try {
       // Original content of methodE
       ...

       // Record a Method Exit event of methodE
   } catch (Throwable e) {
       // Record a Method Exceptional Exit event of methodE
       throw e;
   }
}
```

Fig. 2. An example of logging code locations for recording Method Execution and Call events.

the fields. Similarly, events in the Array Access category record array references and their accessed indices. Although most memory read events just record the same values as their preceding memory write events, the tool records all the Field Access and Array Access events because their values may be updated by library classes whose behavior is not observed by the tool.

Table 2
Events linked to identifiers in source code.

Event	Identifiers in source code
Method Param for a parameter `par`	The formal variable name `par` in a method declaration, e.g. "`void m(Type par)`"
Local Load reading a variable `var`	The variable name `var` in an expression
Local Store writing to a variable `var`	The variable name `var` on the left hand side of an assignment expression, e.g. "`var = `*e*" and "`var += `*e*"
Local Increment updating a variable `var`	(1) The variable name `var` on the left hand side of an assignment expression, e.g. "`var = var + `*constant*" (2) The variable `var` in a pre-/post-increment expression: `var++`, `var-`, `++var`, or `-var`
Get Field Result reading from a field `f`	The field name `f` in an expression
Put Filed Value writing a field `f`	The field name `f` on the left hand side of an assignment expression, e.g. "`f = `*e*" and "`x.f = `*e*"

An execution trace is a sequence of events $\langle d, t, v\rangle$ where d is a *data ID* to identify an event of a particular instruction, t represents the thread that executed the instruction, and v represents the observed value. A data ID d is assigned by the recorder component during the bytecode instrumentation. When the `methodA()` in Fig. 1 is called, the recorder component assigns four IDs to represent the following method execution events.

- A method entry event recording the receiver object (that is, the value of `this`).
- An actual parameter event that records the value of `var` passed to the method.
- A normal exit event. Although the method does not return any value, the event represents a successful termination of the method.
- An exceptional exit event recording an exception if the method is terminated by an exception.

The component also assigns data IDs to every instruction in the method.

Our tool records the latest k events for each data ID d. Then, it takes buffer size k as a parameter, allocates buffers for each data ID, and accumulates the observed events alongside their sequence numbers (i.e., an increasing number representing the order of events). When the program has finished, the tool writes the accumulated data to storage using a shutdown hook function in the Java virtual machine. The maximum trace size is $k \times N$ where N is the number of data elements used by instructions in a program. Our tool with $k = \infty$ is conceptually equivalent to omniscient debugging.

To enable users to investigate how objects are manipulated, our tool translates object references into object IDs composed of its class name and a number. For String and Exception objects, the tool records their textual contents with object IDs for ease of debugging. Our current implementation simply records the textual contents as is. For this reason, in case of long strings, the trace size could still be large. Thus, an effective recording of textual contents will be considered in future work.

3.2. Post processor

The post-processor component links the source code contents of a program to data elements in an execution trace of the program produced by the recorder component. Conceptually, the output of the component is a mapping $\{l \mapsto d\}$, where l is the location of an identifier in the source code and d is a data element in a trace. Source-to-trace mapping enables an interactive view to show appropriate data values for each source code location of interest to a user.

Table 2 shows the runtime events linked to the source code. The component produces one-to-one mapping for those events because a single identifier in the source code corresponds to a method parameter, a local variable, or a field name recorded as those events. For example, suppose an expression `x=y` is in a source file. The assignment `x=` can either be (1) a Local Store event for a local variable `x` or (2) a Put Field Value event for a field `x` corresponding to the line. If those events are recorded, the identifier is linked to them. The link enables an interactive view to show actual values assigned to `x`. Similarly, the variable `y` is linked to either (1) a Local Load event for a local variable `y` or (2) a Get Field Result event for a field `y`. The identifier is linked to those events so that the interactive view can show actual values read from `y`. To link the events and identifiers, the component builds a syntax tree for each source file, extracts identifiers from the tree, and then identifies their corresponding events in the trace. In the last step, the tool searches local variable access and field access events using the source file name, the line number of the expression, and the identifier names such as `x` and `y`. The search depends on the heuristics as shown in Table 2 instead of a deeper semantic analysis of the source code. This is because such a semantic analysis requires the whole program and libraries to take class inheritance and static import mechanisms into account.

Some runtime events have no corresponding tokens in the source code. For example, the return value of a method call may be discarded without an assignment operator. To visualize such invisible events in a trace, the post-processor translates them into pseudo variables as shown in Table 3. The pseudo variables are also included in the source-to-trace mapping.

Table 3
The pseudo variables representing invisible values on source code.

Event Name	Variable Name
Call Return	_ReturnValue
Array Load Result	_ArrayLoad
Array Store Value	_ArrayStore
Array Length Result	_ArrayLength

3.3. Interactive view

The interactive view component is an HTML-based view that works on a web browser. Using the source-to-trace mapping produced by the post-processor component, each tab displays a source file, whose variables are highlighted. By hovering a mouse cursor on a highlighted variable, the actual values of the variable recorded in the trace are displayed. For example, Fig. 3 shows a screenshot of an interactive view displaying the values of `var` in ascending order by time.

This interactive view can filter values by specifying a time interval. For example, in Fig. 3, a user can click on the arrows shown on the right side of each value. A click on the left arrow specifies a start point of an interval, while a click on the right arrow specifies the end point. Fig. 4 shows the interactive view displaying observed values after assigning 32 to `var` at line 15. If no values are recorded for a variable during the specified time interval, the highlighting for that variable is turned off. In Fig. 4, the highlighting of `var` at line 13 disappeared as a result of filtering. It should be noted that interactive views share a single filter within multiple tabs. A user can investigate an inter-procedural data-flow by selecting an arbitrary pair of source code locations in a program.

The interactive view also provides a textual representation of runtime events linked to a source file. The following example shows the textual representation for the example program.

```
...
ID:16, Line:5, Variable:_ReturnValue, Seqnum:45, Data:8
ID:28, Line:10, Variable:var, Seqnum:8, Data:127
...
```

Each line shows five attributes of an event: data ID d (`ID`), line number (`Line`), variable name (`Variable`), sequence number (`Seqnum`), and recorded data value (`Data`). On the textual representation, we can perform a keyword search on the recorded values. We can also refer to the invisible values in a table. For example, we cannot check a returned value from the method call at line 5 because the value is not assigned to any variable. In the above example, we can check the value as a pseudo variable `_ReturnValue`, which means the method returned value on the line. The interactive view uses a subset of events recorded by the recorder component. For example, thread ID is available but omitted from the view. This is because a naive visualization of all the details of runtime events may be too complicated for users.

3.4. Usage

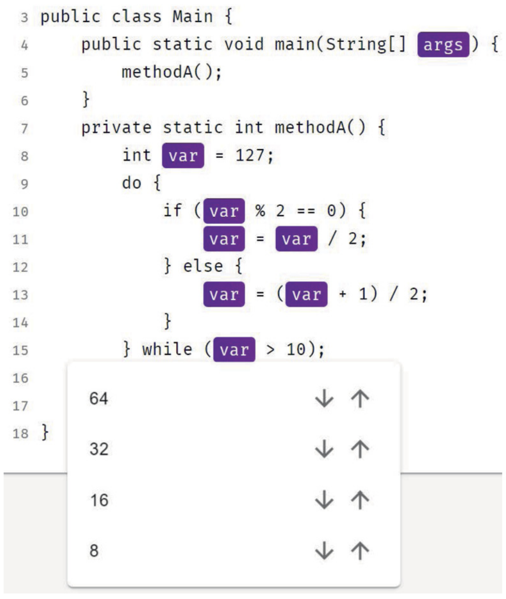
The tool is available on GitHub. The main repository,[2] named `nod4j`, includes the binary files of the tool and the source code of the post-processor and interactive view components. The following command builds the tool from source code using Maven. The `nod4j.jar` file is created in `target` directory.

```
git clone https://github.com/k-shimari/nod4j.git
cd nod4j
mvn package
```

The source code of the recorder component is separated in another repository[3] because the recorder may be used for other research. The tool is dependent on the following tools and libraries. The version numbers show the environment of the authors on Windows 10.

- Java(TM) SE Runtime Environment (build 1.8.0_241-b07)
- Apache Maven (3.6.3)
- Node.js (v12.16.1 LTS)
- npm (6.14.4)

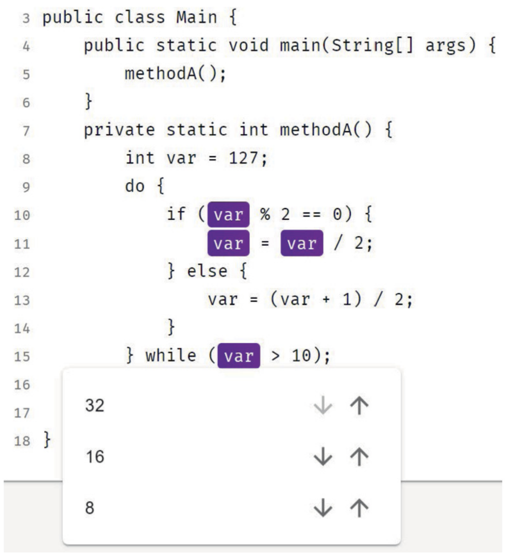
[2] https://github.com/k-shimari/nod4j.
[3] https://github.com/takashi-ishio/selogger.



```
public class Main {
    public static void main(String[] args) {
        methodA();
    }
    private static int methodA() {
        int var = 127;
        do {
            if (var % 2 == 0) {
                var = var / 2;
            } else {
                var = (var + 1) / 2;
            }
        } while (var > 10);
}
```

```
64
32
16
8
```

Fig. 3. Trace View.



```
public class Main {
    public static void main(String[] args) {
        methodA();
    }
    private static int methodA() {
        int var = 127;
        do {
            if (var % 2 == 0) {
                var = var / 2;
            } else {
                var = (var + 1) / 2;
            }
        } while (var > 10);
}
```

```
32
16
8
```

Fig. 4. Trace View with Filtering.

To explain the usage of our tool, we performed a debugging session of a small program. This program is included in the `sample/demo/for_build` directory of the `nod4j` repository. The program implements a method to select the maximum number from the three given numbers as follows:

```
public class Main {
    public static int getMax(int num1, int num2, int num3) {
        // return the maximum number of three arguments
    }
}
```

To test the method, the file `getMaxTest.java` contains the following test cases that provide parameters 10, 20, and 30 in different orders to the target method.

```
10:        @Test
11:        public void getMaxTest1() {
12:            assertEquals(30, Main.getMax(30, 10, 20));
13:            assertEquals(30, Main.getMax(30, 20, 10));
14:            assertEquals(30, Main.getMax(20, 10, 30));
15:            assertEquals(30, Main.getMax(20, 30, 10));
16:            assertEquals(30, Main.getMax(10, 20, 30));
17:            assertEquals(30, Main.getMax(10, 30, 20));
18:        }
```

The following command executes the test cases of the sample program.

```
cd sample/demo/for_build/
mvn test
```

The command reports a failure as follows:

```
java.lang.AssertionError: expected:<30> but was:<20>
    at testsample.getMaxTest.getMaxTest1(getMaxTest.java:17)
```

The last test case at line 17 fails and a log message for the test case shows that the expected return value is 30 but the actual return value is 20.

To debug the problem using our tool, the `pom.xml` file for the program specifies the following argument for a Java VM executing the test case.

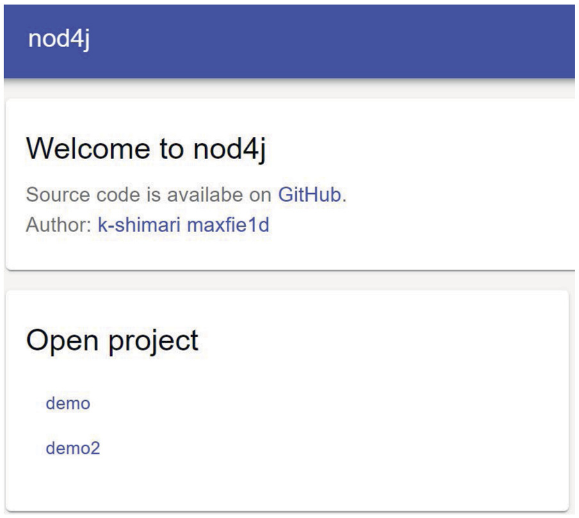

nod4j
Welcome to nod4j
Source code is availabe on GitHub.
Author: k-shimari maxfie1d
Open project
demo
demo2

Fig. 5. The Welcome Page of NOD4J.

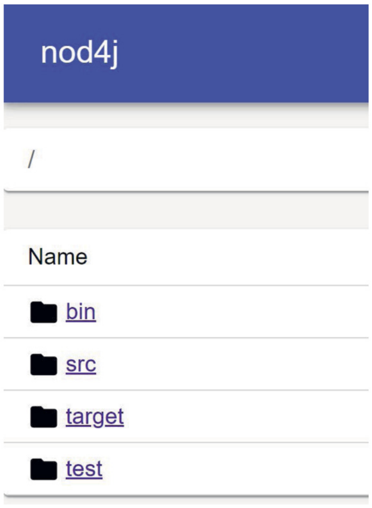


Fig. 6. The Browser Page of NOD4J.

```
-javaagent:"../../../selogger-0.2.3.jar=output=../selogger,size=64,
e=junit/framework/,e=org/junit/,e=org/apache/maven,e=org/hamcrest"
```

The `-javaagent` option points to the recorder component called `selogger-0.2.3.jar` located in the top directory of the main repository. The remaining text is the parameters for recording. The `output=../selogger` parameter specifies an output directory. The `size=64` parameters specify the buffer size $k = 64$. The "`e=...`" parameters specify package names to be excluded from the execution trace. In this configuration, the recorder component excludes the Maven and JUnit classes. The execution trace is stored in the `sample/demo/selogger` directory.

To use the interactive view, we need to link the source code to the execution trace.

```
java -jar nod4j.jar sample/demo/for_build sample/demo/selogger
  src/main/frontend/src/assets/project/demo
```

The three arguments specify a source code directory, an execution trace directory, and an output directory, respectively.

The command produces two json files, `fileinfo.json` and `varinfo.json`, in the specified output directory. We put the json files in `src/main/frontend/src/assets/project/demo` so that the interactive view can read the json files. Then, we can build HTML contents from the json files and start a server for the contents. To build HTML contents, we execute the following commands in the `src/main/frontend` directory.

```
npm install
npm run build
npm run server
```

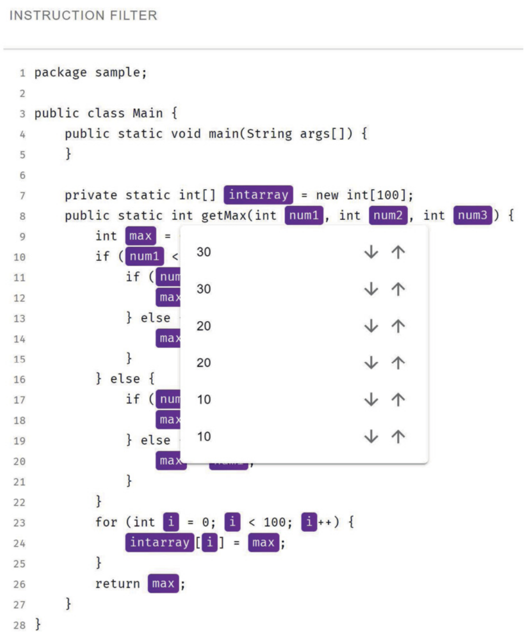
The `npm install` command is for installing dependencies, so this command needs to be run only once. The `npm run build` command executes the post-processor component. The `npm run server` command starts a web server for the interactive view. A web browser can access the view through a local address `localhost:8070`.

Fig. 5 shows the welcome page of the interactive view. NOD4J checks subdirectories in `src/main/frontend/src/assets/project/demo` and automatically shows the names in the project list. Then, by clicking the project name and traversing the directory shown in Fig. 6, we can access the file we want to see the execution.

To analyze the failure of the example program, open the file `src/sample/Main.java`. Fig. 7 shows the view of the target method in the file. All test cases return a value on the variable `max` at line 26. Then, we can confirm that the sixth return value, 20, caused the test failure. To visualize the computation for this invalid return value, we filter the execution with a time interval from the initialization of `num1` at line 8 by clicking on the left arrow of the sixth value of `num1` at line 8.

Fig. 8 shows the filtering result. The highlighted variables show that the `max` was incorrectly assigned at line 12. We can fix this bug by changing the variable `num1` to `num2` at line 11.

In this example, the target method includes a loop at lines 23–25 that is irrelevant to the test cases. Although the loop results in a large number of steps in a full execution trace, our tool excludes those steps from the trace.



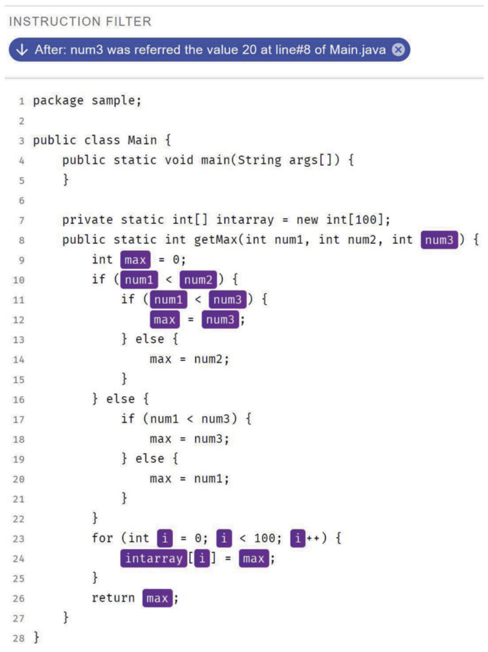
```
package sample;

public class Main {
    public static void main(String args[]) {
    }

    private static int[] intarray = new int[100];
    public static int getMax(int num1, int num2, int num3) {
        int max =
        if (num1 <
            if (num
                max
            } else
                max
            }
        } else {
            if (num
                max
            } else
                max
            }
        }
        for (int i = 0; i < 100; i++) {
            intarray[i] = max;
        }
        return max;
    }
}
```

```
30
30
20
20
10
10
```

Fig. 7. The Interactive View for the Current Scenario.


INSTRUCTION FILTER
After: num3 was referred the value 20 at line#8 of Main.java

```
package sample;

public class Main {
    public static void main(String args[]) {
    }

    private static int[] intarray = new int[100];
    public static int getMax(int num1, int num2, int num3) {
        int max = 0;
        if (num1 < num2) {
            if (num1 < num3) {
                max = num3;
            } else {
                max = num2;
            }
        } else {
            if (num1 < num3) {
                max = num3;
            } else {
                max = num1;
            }
        }
        for (int i = 0; i < 100; i++) {
            intarray[i] = max;
        }
        return max;
    }
}
```

Fig. 8. The Result of the Filtering.

4. Usage examples

We illustrate two actual debugging sessions using the tool. We use bugs in the Defects4J benchmark version 1.5.0[4] executed on Java(TM) SE Runtime Environment (build 1.7.0_80-b15). We chose Lang 2b and Math 59b as examples, because the former resulted in a relatively smaller execution trace among the bugs and the latter resulted in a larger execution trace. We recorded the execution trace of a failed test method for each bug. When more than one test method failed, we ran only the top one shown in the result of the `detects4j info` command. The traces record only the behavior of the source code of the project. In other words, the recorded traces contain method calls to libraries (e.g., `Assertion` class in JUnit) in the target unit tests but do not contain the internal behavior of the libraries.

4.1. Use Case 1: Lang 2b

The first example is Lang 2b, a bug of the `LocaleUtils.toLocale()` method. The method takes a `String` object representing a locale name and returns a `Locale` object corresponding to the name if the name is in the correct format. The method is tested using a method named `testParse AllLocales()` in the file `src/test/java/org/apache/commons/lang3/LocaleUtilsTest.java`. The test method fails and provides the following log messages to the standard output:

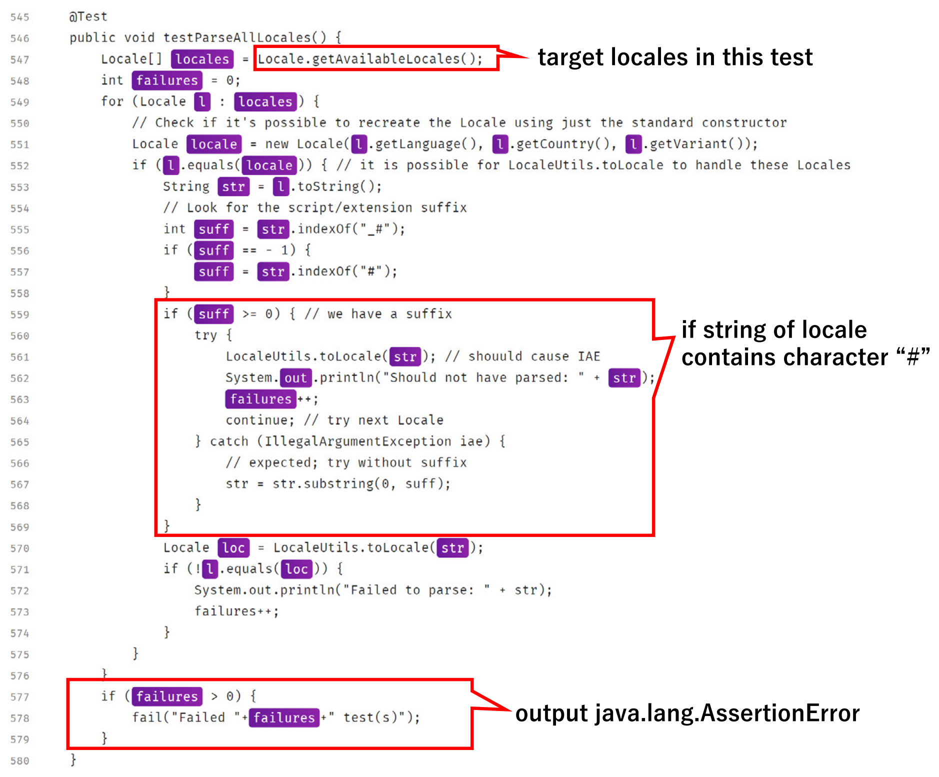
```
    Should not have parsed: ja_JP_JP_#u-ca-japanese
    Should not have parsed: th_TH_TH_#u-nu-thai
```

To debug this problem, we obtained an execution trace of the failed test method using our trace recorder with $k = 64$. The resulting execution trace completely recorded the behavior of 262 out of 418 instructions (62.7%). The other instructions were executed more than 64 times. The size of the trace is 167 KB, which is 52.2% of its full execution trace (348 KB).

Fig. 9 shows the test method `testParseAllLocales()` in the interactive view. The test method obtains all available locales and passes their names to the method under test. If the locale name has a suffix (“#”), the method under test should throw an exception. Otherwise, the method should return a `Locale` object corresponding to the locale name. The test method counts the number of occurrences of incorrect behavior using a variable `failures`. The test method failed because the `failures` was greater than zero.

The interactive view provides the actual values of the `str` variable as shown in Fig. 10. From this information, we can confirm that this test failed because the method under test did not throw `IllegalArgumentException` for locale

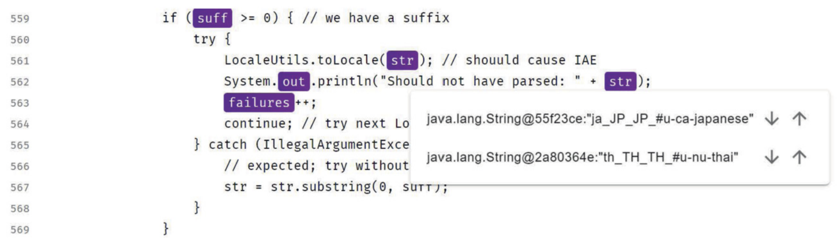
[4] https://github.com/rjust/defects4j/, Commit `bd32d9642e12`.



```
545     @Test
546     public void testParseAllLocales() {
547         Locale[] locales = Locale.getAvailableLocales();
548         int failures = 0;
549         for (Locale l : locales) {
550             // Check if it's possible to recreate the Locale using just the standard constructor
551             Locale locale = new Locale(l.getLanguage(), l.getCountry(), l.getVariant());
552             if (l.equals(locale)) { // it is possible for LocaleUtils.toLocale to handle these Locales
553                 String str = l.toString();
554                 // Look for the script/extension suffix
555                 int suff = str.indexOf("_#");
556                 if (suff == - 1) {
557                     suff = str.indexOf("#");
558                 }
559                 if (suff >= 0) { // we have a suffix
560                     try {
561                         LocaleUtils.toLocale(str); // shouuld cause IAE
562                         System.out.println("Should not have parsed: " + str);
563                         failures++;
564                         continue; // try next Locale
565                     } catch (IllegalArgumentException iae) {
566                         // expected; try without suffix
567                         str = str.substring(0, suff);
568                     }
569                 }
570                 Locale loc = LocaleUtils.toLocale(str);
571                 if (!l.equals(loc)) {
572                     System.out.println("Failed to parse: " + str);
573                     failures++;
574                 }
575             }
576         }
577         if (failures > 0) {
578             fail("Failed "+failures+" test(s)");
579         }
580     }
```

target locales in this test
if string of locale contains character "#"
output java.lang.AssertionError

Fig. 9. A test method for `LocaleUtils.toLocale` method in Lang 2b.



```
559                 if (suff >= 0) { // we have a suffix
560                     try {
561                         LocaleUtils.toLocale(str); // shouuld cause IAE
562                         System.out.println("Should not have parsed: " + str);
563                         failures++;
564                         continue; // try next Lo
565                     } catch (IllegalArgumentExce
566                         // expected; try without
567                         str = str.substring(0, suff);
568                     }
569                 }
```

```
java.lang.String@55f23ce:"ja_JP_JP_#u-ca-japanese"
java.lang.String@2a80364e:"th_TH_TH_#u-nu-thai"
```

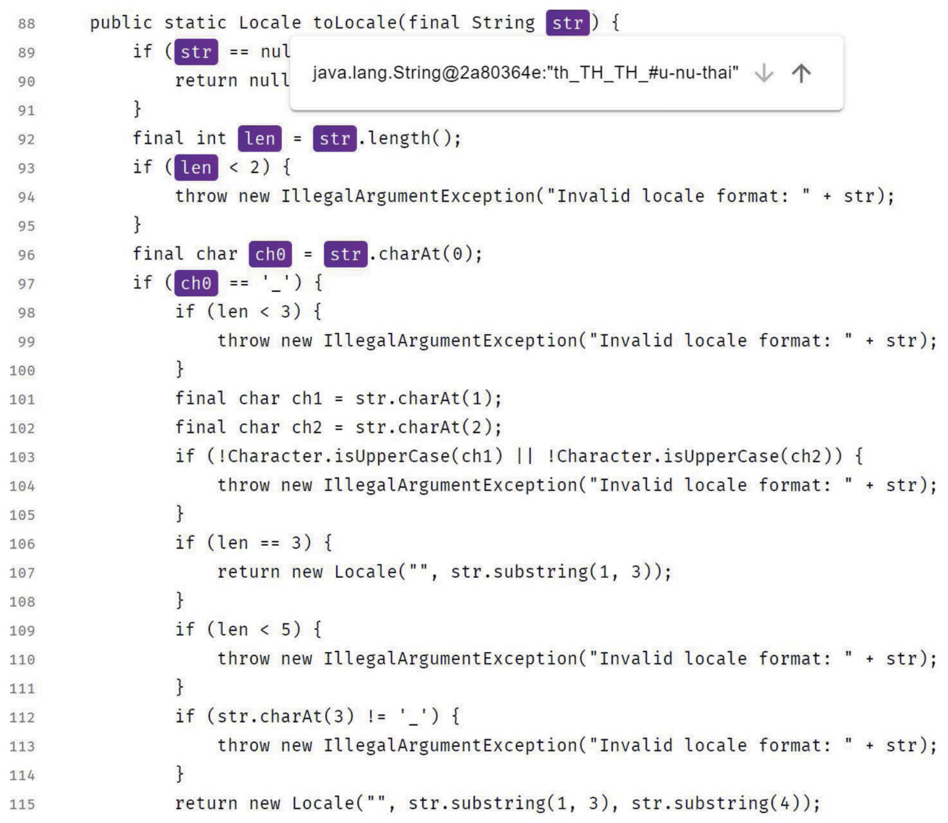
Fig. 10. Actual parameter values that induced a failure in Lang 2b.

names including "#". Fig. 9 also shows that lines 572 through 573 were not executed because the variables on the lines are not highlighted.

Then, we open the file that contains the target method `toLocale()` to analyze the failure in detail. The first half of the method is shown in Fig. 11, the second half is in Fig. 12, respectively. While the method is executed several times, we are interested in only a failed execution whose argument includes "#". Hence, we filter the interesting behavior by clicking on a down arrow on the right side of a string value "`th_TH_TH_#u-nu-thai`" shown in Fig. 11. This view shows a partial trace recorded after the value was observed at the beginning of the method. The highlighted variables show the execution flow at that time. In Fig. 11, there is no highlighted variable in the `if` statement's body from line 98 to 115. However, there are many highlighted variables from line 117 to 147 in Fig. 12. Then, we can see that this method call executes `else` statement's body and returns on line 147. At line 147, we can find that the method passed a result of a method call "`str.substring(6)`" to a constructor of `Locale` class. The `if` statements do not check "#" at all in the method. This is the cause of the bug. In the fixed version of the program, the following `if` block is inserted into the method at line 92.

```
if (str.contains("#")) {
    throw new IllegalArgumentException("Invalid locale format: " + str);
}
```

In this debugging session, the actual values of variables recorded in the trace are effective to investigate the behavior of the failed test. While the trace does not record the complete execution, the trace still maintains variable values observed for the corner case.



```
public static Locale toLocale(final String str) {
    if (str == nul
        return null
                        java.lang.String@2a80364e:"th_TH_TH_#u-nu-thai"
    }
    final int len = str.length();
    if (len < 2) {
        throw new IllegalArgumentException("Invalid locale format: " + str);
    }
    final char ch0 = str.charAt(0);
    if (ch0 == '_') {
        if (len < 3) {
            throw new IllegalArgumentException("Invalid locale format: " + str);
        }
        final char ch1 = str.charAt(1);
        final char ch2 = str.charAt(2);
        if (!Character.isUpperCase(ch1) || !Character.isUpperCase(ch2)) {
            throw new IllegalArgumentException("Invalid locale format: " + str);
        }
        if (len == 3) {
            return new Locale("", str.substring(1, 3));
        }
        if (len < 5) {
            throw new IllegalArgumentException("Invalid locale format: " + str);
        }
        if (str.charAt(3) != '_') {
            throw new IllegalArgumentException("Invalid locale format: " + str);
        }
        return new Locale("", str.substring(1, 3), str.substring(4));
```

Fig. 11. A filtering result of the `toLocale` method in Lang 2b (The first half of the method).

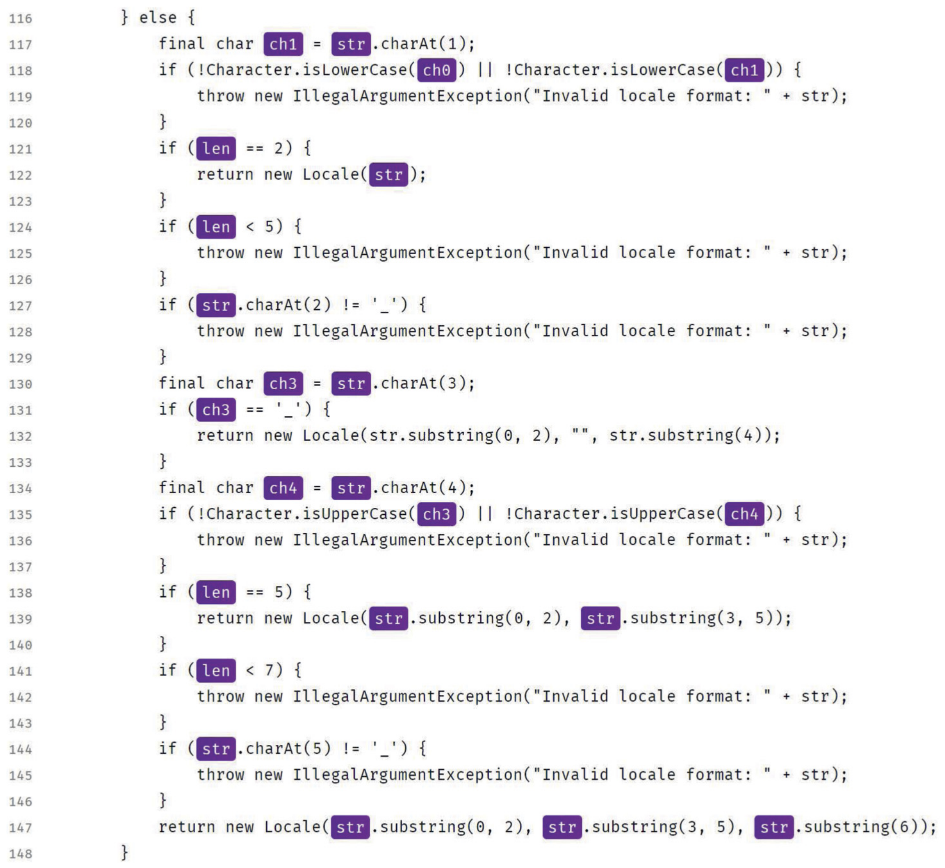
4.2. *Use Case 2: Math 59b*

The second example is Math 59b. The method `FastMath.max()` compares two `float` values and returns a greater one. The method is tested by the test method `testMinMaxFloat()` in the source file `src/test/java/org/apache/commons/math/util/FastMathTest.java`. This test method calls the `FastMath.max()` method with various parameters and compares the results with `Math.max()` as follows.

```
 78:  public void testMinMaxFloat(){
 79:    float[][] pairs = {
 80:        { -50.0f, 50.0f },
        ...
 89:    };
 90:    for(float[] pair : pairs){
        ...
 99:      Assert.assertEquals("max(" + pair[0] + "," + pair[1] + ")",
100:                          Math.max(pair[0],pair[1]),
101:                          FastMath.max(pair[0],pair[1]),
102:                          Math.Utils.EPSILON);
103:      Assert.assertEquals("max(" + pair[1] + "," + pair[0] + ")",
104:                          Math.max(pair[1],pair[0]),
105:                          FastMath.max(pair[1],pair[0]),
106:                          Math.Utils.EPSILON);
107:    }
108:  }
```

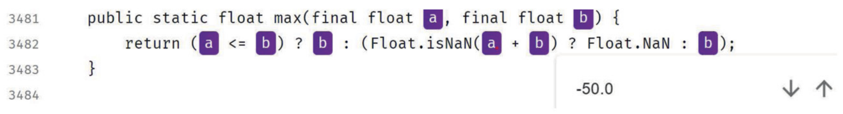
The test method fails and produces the following message.

```
java.lang.AssertionError: max(50.0, -50.0) expected:<50.0> but was:<-50.0>
at org.apache.commons.math.util.FastMathTest.testMinMaxFloat (FastMathTest.java:103)
```



```
} else {
    final char ch1 = str.charAt(1);
    if (!Character.isLowerCase(ch0) || !Character.isLowerCase(ch1)) {
        throw new IllegalArgumentException("Invalid locale format: " + str);
    }
    if (len == 2) {
        return new Locale(str);
    }
    if (len < 5) {
        throw new IllegalArgumentException("Invalid locale format: " + str);
    }
    if (str.charAt(2) != '_') {
        throw new IllegalArgumentException("Invalid locale format: " + str);
    }
    final char ch3 = str.charAt(3);
    if (ch3 == '_') {
        return new Locale(str.substring(0, 2), "", str.substring(4));
    }
    final char ch4 = str.charAt(4);
    if (!Character.isUpperCase(ch3) || !Character.isUpperCase(ch4)) {
        throw new IllegalArgumentException("Invalid locale format: " + str);
    }
    if (len == 5) {
        return new Locale(str.substring(0, 2), str.substring(3, 5));
    }
    if (len < 7) {
        throw new IllegalArgumentException("Invalid locale format: " + str);
    }
    if (str.charAt(5) != '_') {
        throw new IllegalArgumentException("Invalid locale format: " + str);
    }
    return new Locale(str.substring(0, 2), str.substring(3, 5), str.substring(6));
}
```

Fig. 12. A filtering result of the `toLocale` method in Lang 2b (The second half of the method).



```
3481    public static float max(final float a, final float b) {
3482        return (a <= b) ? b : (Float.isNaN(a + b) ? Float.NaN : b);
3483    }
3484
```

Fig. 13. Failure induced method at Math 59b.

The method under test returned an incorrect value -50.0 for the first value (`-50.0f, 50.0f`), written in line 80.

We recorded an execution trace of the test method with $k = 64$. It resulted in 4.0 MB. 3,841 out of 7,356 instructions (47.8%) were executed more than 64 times; the full execution trace of the test method resulted in 12.4 GB. Our tool recorded only 0.03% of runtime events. The cause of the significant reduction is the setting up of method defined as follows.

```
@Before
public void setUp() {
  field = new DfpField(40);
  generator = new MersenneTwister(6176597458463500194l);
}
```

Before the unit test is executed, the setup method is executed. The method is short but executes a large number of methods; all the runtime events excluded from our trace belong to the method. The recorded trace includes the complete execution of the test method.

The test targets are `FastMath.min()` and `FastMath.max()`, which return the minimum and maximum values of the two values in the file `src/main/java/org/apache/commons/math/util/FastMath.java`. From the error message, we have to check the execution trace of `FastMath.max()`. Fig. 13 shows a view of the method. We can easily see the actual values used in the two method calls. Then, we can easily find that we should fix the value of the last variable *b* to *a* at line 3,482. This case also shows that a complete execution trace is not always needed for debugging.

5. Conclusion

NOD4J monitors and visualizes detailed software behavior with reducing storage space consumption. The tool is suitable for monitoring remote program execution, such as testing on continuous integration servers, and visualizing the behavior of test failures on a web browser. As illustrated in the usage examples, our tool can debug defects using incomplete execution traces, while it can record complete execution traces for many bugs.

In future work, we would like to investigate effective logging for textual content such as strings and exceptions. Another future research direction is the development of automated debugging methods utilizing near-omniscient execution trace. We also would like to improve the interactive view page so that it can provide an understandable view for large values of k with visualizing the sequence of the values of variables. Finally, as part of our future work, we will ensure that NOD4J responds dynamically to added/removed execution traces.

CRediT authorship contribution statement

Kazumasa Shimari: Data curation, Investigation, Methodology, Software, Visualization, Writing – original draft. **Takashi Ishio:** Conceptualization, Methodology, Software, Writing – review & editing. **Tetsuya Kanda:** Validation, Writing – review & editing. **Naoto Ishida:** Software. **Katsuro Inoue:** Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work has been supported by JSPS KAKENHI Nos. JP18H03221, JP18KT0013, JP18H04094 and JP19K20239.

References

[1] D. Spinellis, Effective Debugging: 66 Specific Ways to Debug Software and Systems, Addison-Wesley Professional, 2016.
[2] H. Cleve, A. Zeller, Locating causes of program failures, in: Proc. ICSE, 2005, pp. 342–351.
[3] N.M. Johnson, J. Caballero, K.Z. Chen, S. McCamant, P. Poosankam, D. Reynaud, D. Song, Differential slicing: identifying causal execution differences for security applications, in: Proc. of the 32nd IEEE Symposium on Security and Privacy, 2011, pp. 347–362.
[4] P.V. Gestwicki, B. Jayaraman, JIVE: Java interactive visualization environment, in: Companion Proc. OOPSLA, 2004, pp. 226–228.
[5] S. Kabinna, C.-P. Bezemer, W. Shang, A.E. Hassan, Logging library migrations: a case study for the apache software foundation projects, in: Proc. MSR, 2016, pp. 154–164.
[6] D. Yuan, J. Zheng, S. Park, Y. Zhou, S. Savage, Improving software diagnosability via log enhancement, SIGARCH Comput. Archit. News 39 (1) (2011) 3–14.
[7] B. Lewis, Debugging backwards in time, CoRR, arXiv:cs/0310016 [cs.SE], 2003.
[8] G. Pothier, E. Tanter, J. Piquer, Scalable omniscient debugging, in: Proc. OOPSLA, 2007, pp. 535–552.
[9] B. Cornelissen, L. Moonen, A. Zaidman, An assessment methodology for trace reduction techniques, in: Proc. ICSM, 2008, pp. 107–116.
[10] K. Shimari, T. Ishio, T. Kanda, K. Inoue, Near-omniscient debugging for Java using size-limited execution trace, in: Proc. ICSME, 2019, pp. 398–401.
[11] G.W. Dunlap, S.T. King, S. Cinar, M.A. Basrai, P.M. Chen, ReVirt: enabling intrusion analysis through virtual-machine logging and replay, SIGOPS Oper. Syst. Rev. (2003) 211–224.
[12] N. Honarmand, J. Torrellas, Replay debugging: leveraging record and replay for program debugging, in: Proc. ISCA, 2014, pp. 455–456.
[13] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, N. Partush, Engineering record and replay for deployability, in: Proc. USENIX ATC, 2017, pp. 377–389.
[14] T. Wang, A. Roychoudhury, Using compressed bytecode traces for slicing Java programs, in: Proc. ICSE, 2004, pp. 512–521.
[15] M. Hirzel, T. Chilimbi, Bursty tracing: a framework for low-overhead temporal profiling, in: Proc. the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization, 2001.
[16] S. Mirghasemi, J.J. Barton, C. Petitpierre, Querypoint: moving backwards on wrong values in the buggy execution, in: Proc. ESEC/FSE, 2011, pp. 436–439.
[17] T. Matsumura, T. Ishio, Y. Kashima, K. Inoue, Repeatedly-executed-method viewer for efficient visualization of execution paths and states in Java, in: Proc. ICPC, 2014, pp. 253–257.
[18] E. Hilsdale, J. Hugunin, Advice weaving in AspectJ, in: Proc. the 3rd International Conference on Aspect-Oriented Software Development, 2004, pp. 26–35.