Repeatedly-Executed-Method Viewer for Efficient Visualization of Execution Paths and States in Java

Toshinori Matsumura, Takashi Ishio, Yu Kashima, Katsuro Inoue
Osaka University
1-5 Yamadaoka, Suita, Osaka 565-0871, Japan
{m-tosinr, ishio, y-kasima, inoue}@ist.osaka-u.ac.jp

ABSTRACT

The state of a program at runtime is useful information for developers to understand a program. Omniscient debugging and logging-based tools enable developers to investigate the state of a program at an arbitrary point of time in an execution. While these tools are effective to analyze the state at a single point of time, they might be insufficient to understand the generic behavior of a method which includes various control-flow paths. In this paper, we propose REMViewer (Repeatedly-Executed-Method Viewer), or a tool that visualizes multiple execution paths of a Java method. The tool shows each execution path in a separated view so that developers can firstly select actual execution paths of interest and then compare the state of local variables in the paths.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Tracing*

General Terms

Experimentation

Keywords

dynamic analysis, program understanding, record-and-replay, Java

1. INTRODUCTION

To understand a program, developers often execute a program and analyze the state of the program at runtime using a debugger [10, 12]. While traditional interactive debuggers including GDB and Eclipse JDT are widely used, such tools require developers to explore source code in advance to set breakpoints of potential interest [9].

Omniscient debugging and logging-based techniques [6, 7, 8] are promising approaches to understanding the dynamic behavior of a program for software maintenance. Omniscient debugging [8] enables developers to inspect the state of a program at an arbitrary point of time in an execution, by recording all the runtime events during the execution of the program. Although recording an execution makes a program slower, it removes the burden of prior source code exploration.

Since the existing techniques are proposed for debugging, they focus on only a single point of time in an execution. While investigating a particular point of time related to a bug is sufficient for debugging [7], it may be insufficient to understand the generic behavior of a method. If a method includes various control-flow paths to implement a complicated functionality, developers have to analyze individual control-flow paths in the method [11].

In this paper, we propose REMViewer (**R**epeatedly-**E**xecuted-**M**ethod **Viewer**), or a tool that visualizes multiple executions of a Java method. While a method may contain branches to handle various special cases, most of executions of the method may pass through a small number of control-flow paths representing normal cases. Hence, REMViewer classifies executions of a method into groups according to the executed instructions; it picks up an execution from each group as a representative for visualization. By comparing visualized paths, developers can firstly select actual execution paths of interest and then analyze the states of local variables at each line in the paths.

The remaining of the paper is organized as follows. Section 2 explains the features of our tool. Section 3 shows the performance of the tool. Section 4 describes the related work. In Section 5, we present the conclusion and future work.

2. TOOL FEATURES

REMViewer is a logging-based visualization tool for Java. The tool records an execution of a program and replays the behavior of a method selected by a user. The tool comprises two components: *instrumentation* and *replay*. The instrumentation component inserts logging instructions into class files of a target program. An execution of the instrumented program records an execution trace. The replay component recovers the runtime states of execution paths of a method selected by a user. A user can browse the recovered behavior using a graphical user interface.

2.1 Execution Trace

An execution trace for REMViewer is a sequence of runtime events representing control-flow and inter-procedural data-flow. Control-flow events include method entry, method exit, method call, and exception handling. Inter-procedural

ICPC '14, June 2-3, 2014, Hyderabad, India

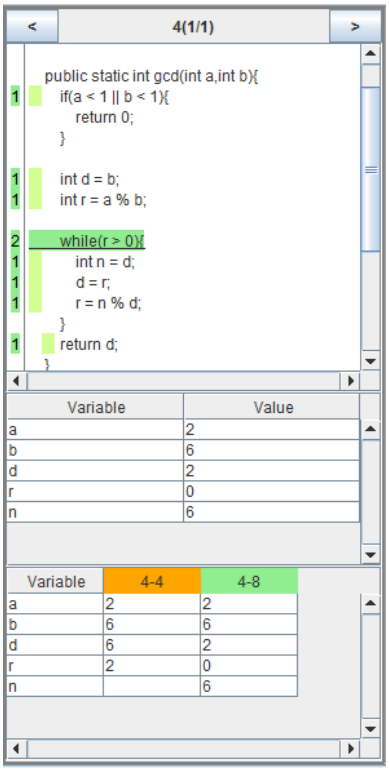

Figure 1: A path view that shows an execution of a target method

data-flow events include field access and array access.

Every event has two common attributes: thread ID and bytecode instruction ID. They identify a thread and a bytecode instruction where an event is observed. In addition, each event has actual values used in the event. For example, when a `PUTFIELD` instruction updated a field of an object with a new value, the ID of the target object and the new field value are recorded. Object ID is sequentially assigned for each object during the execution.

Multi-threaded behavior of a program is recorded as a single sequence of runtime events. If two or more events are observed at the same time, these events are serialized by a `synchronized` block in a logging process. Hence, the logging process may affect thread scheduling in a target program.

An execution trace is stored in a sequence of files. Since an execution of a program may result in a huge number of events, a trace is split into files and compressed by the `gzip` algorithm. To reduce the runtime overhead, a thread for logging is created in addition to application threads. After an execution is terminated, REMViewer makes an index of method entry events so that the replay component can replay a method from those events.

2.2 Replay

The replay component recovers the state of a selected method by interpreting bytecode instructions using the actual values recorded in the trace. The replayed executions of a method is represented by $E = \{e_1, \cdots, e_{|E|}\}$. An execution of a method e_i is represented by a sequence of program

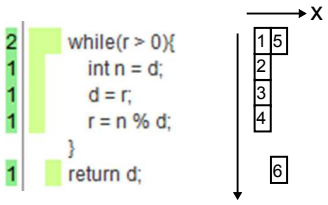

Figure 2: A path represented by rectangles

states $\langle b_k, l_k, v_k \rangle$, where b_k is the k-th executed bytecode instruction, l_k is a line number corresponding to b_k, and v_k is the state of local variables before the execution of b_k.

REMViewer provides a *path view* for an execution path e as shown in Figure 1. The view shows source code in the top. Similarly to a debugger, the view has a cursor pointing to an instruction in an execution. A user can move the cursor forward and backward according to e. The state of variables at the pointed instruction is shown as a table in the middle of the view. In addition, a user can click on another line in source code to inspect the runtime states at the line. A table in the bottom of the view shows the states of variables at the selected line. Since a line in a loop may be executed multiple times, all the states are listed as columns. In the case of Figure 1, two columns show the states of variables at the first and the second execution of the predicate of the selected `while` statement.

To visualize the execution path in a view, green rectangles are drawn in the background of source code. As shown in Figure 2, each instruction $\langle b_k, l_k, v_k \rangle$ in an execution e is represented by a rectangle at a position (x_k, l_k). The horizontal position is incremented when an execution moves upwards in source code; *i.e.* $x_k = x_{k-1} + 1$ if $l_{k-1} > l_k$, otherwise $x_k = x_{k-1}$.

REMViewer opens multiple path views so that developers can compare multiple executions of a method. Because visualizing executions that passed through the same instructions is not so informative for developers, REMViewer automatically classifies the executions E for a method into *path groups*. We have employed two classification: path-based and line-based. The path-based classification classifies a set of executions E into $P(E) = \{P_1, \cdots, P_{|P(E)|}\}$ such that

$$\forall e_1 \in P_i, e_2 \in P_j. \quad i = j \Leftrightarrow Path(e_1) = Path(e_2)$$

where $Path(e)$ is a sequence of bytecode instructions (b_1, $b_2, \cdots$) executed in e. Similarly, the line-based classification results in $L(E) = \{L_1, \cdots, L_{|L(E)|}\}$ such that

$$\forall e_1 \in L_i, e_2 \in L_j. \quad i = j \Leftrightarrow Lines(e_1) = Lines(e_2)$$

where $Lines(e)$ is a set of line numbers corresponding to bytecode instructions executed in e. REMViewer picks up an execution path from each path group as a representative of the group.

Figure 3 shows a screenshot of REMViewer that involves three path views for the method shown in Figure 1. The left view shows an execution that passed through `false` branches for both the `if` and `while` statements in the method. The middle one returns from the method in the `if` statement. The right one executed the body of the `while` statement once. Although the body of the `while` statement could be executed a number of times, these three views cover the basic

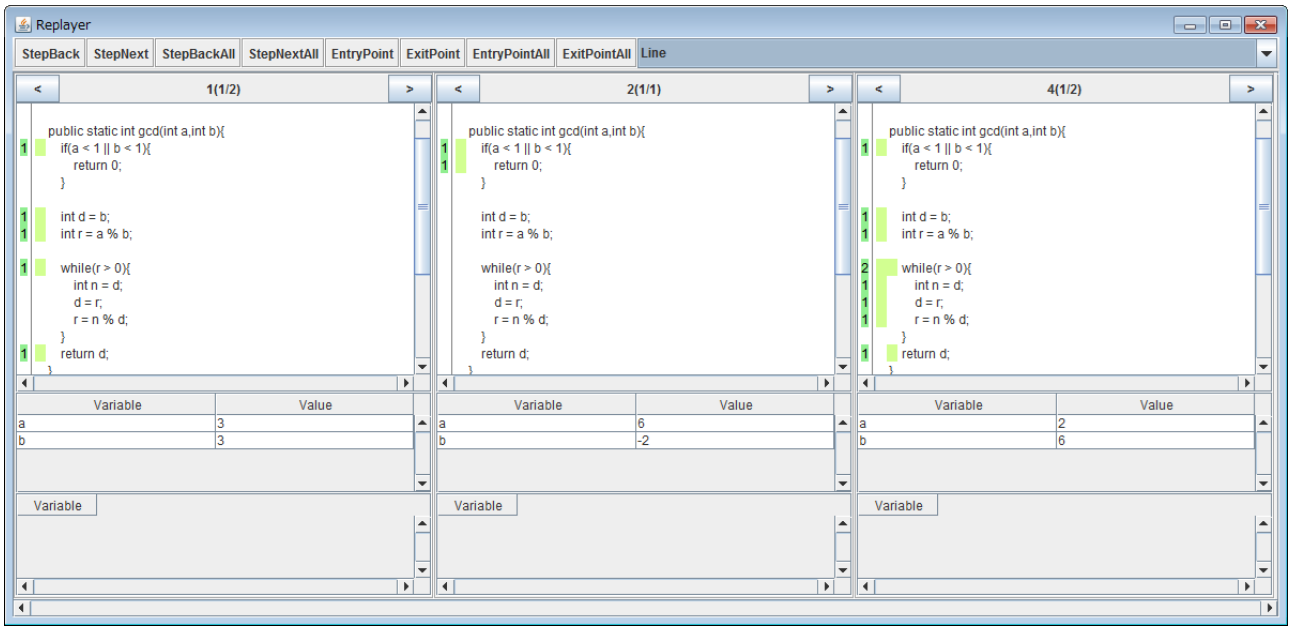

Figure 3: A screenshot of REMViewer that shows three execution paths

Table 1: Execution Trace

Benchmark	`batik`	`fop`
Time (Normal)	4.44 sec	2.80 sec
Time (Logging)	33.67 sec	36.91 sec
Trace Size	717MB	860MB
#Methods	3,131	3,611
#Method-Executions	17,319,017	26,527,278

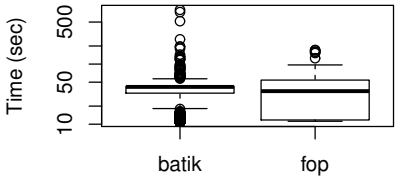

Figure 4: Time for replaying a method

behavior of the method. A user can compare execution paths and the states of variables in a single window.

3. PERFORMANCE EVALUATION

We have evaluated the tool with respect to two perspectives: execution time and visualization applicability. We have used two target applications `batik` and `fop` in the DaCapo benchmark suite 9.12-bach [3]. We have executed their `default` scenarios and recorded the behavior of classes included in the DaCapo binary files. The runtime environment is a workstation equipped with Intel Xeon 2.90GHz.

3.1 Execution Time

Table 1 shows the trace information. The row `Time (Normal)` shows the time for a normal execution without logging. `Time (Logging)` shows the time to execute a benchmark with logging. `Trace Size` is the total size of trace files. `#Methods` indicates the number of executed methods in a program. `#Method-Executions` indicates the total number of method executions.

Logging significantly makes the programs slower, because we have recorded all the classes in the applications. If developers are interested in particular components in a program, they can reduce the logging overhead by recording only the behavior of the selected components.

Figure 4 shows the distribution of the time required for replaying and visualizing a method. Replaying and visualizing a method took 37.5 seconds for `batik` and 36.1 seconds for `fop` on average. Except for 7 methods in `batik`, replay for a method took less than 180 seconds. Hence, our visualization is applicable to most of the methods in the programs. One of the most time-consuming methods is the `runBenchmark` method in `org.dacapo.harness.TestHarness` class. It took 733 seconds, partly because of our naive data structure. Most of the time are spent to process the events between a method entry nearby the beginning of a trace and its corresponding method exit nearby the end of the trace.

3.2 Path Classification

We have applied both path-based and line-based classification to every method in the programs. Because methods without conditional branches are always classified into one group, we have evaluated 1,055 methods in `batik` and 1,243 methods in `fop` that contain at least one branch. Each of those methods contains 4.36 branches and has been executed 9,217 times on average.

Figure 5 shows a cumulative relative frequency graph of the number of classified path groups for each classification. The graph shows 60% of methods are classified into one group by the path-based classification; in other words, only

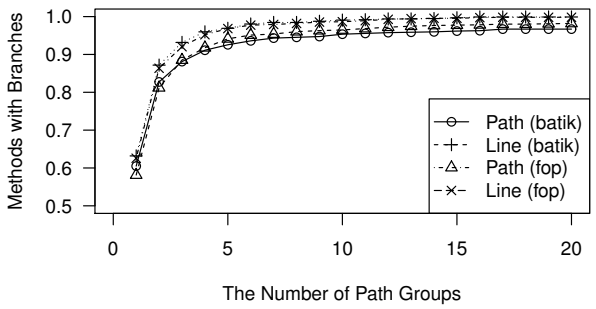

Figure 5: A cumulative relative frequency graph of the number of path groups

one path is executed for those methods. 90% of methods are visualized by at most three views by the line-based classification and five views by the path-based classification. This result shows REMViewer enables developers to investigate a few actual paths out of possible control-flow paths in source code.

On the other hand, the largest number of line-based groups for a method is 30. The method `considerLegalBreak` in `org.apache.fop.layoutmgr.BreakingAlgorithm` class contains 28 branches. To analyze such a complicated method, we would like to improve our path classification technique. For example, automatically classifying execution paths of a method into normal (frequent) cases and special (less frequent) cases might enable developers to start their investigation from the normal behavior of the method.

4. RELATED WORK

Replaying a single method is a selective capture and replay technique proposed by Joshi and Orso [6]. Our tool offers a new user interface for analyzing the behavior of a method.

Jones *et al.* [5] proposed a tool named Tarantula to compare the instructions executed by a number of test cases. Our tool provides classified paths and actual data used in executions for understanding, instead of faulty statements for debugging. Abramson *et al.* [1] proposed Relative Debugging. While it compares an execution of a program with an execution of a different version of the program, our tool compares multiple executions of a single method.

Bell *et al.* [2] proposed Chronicler to record execution traces with low overhead. The performance of our tool might be improved by adopting the approach.

Dynamic invariant detection [4] extracts invariants from actual values of variables in an execution trace. Sagdeo *et al.* [11] proposed a predicate clustering for extracting invariants for a particular set of paths in a method. While our tool simply makes a list of actual values of local variables, the technique is expected to summarize the states of variables in thousands of executions.

5. CONCLUSION

We have presented REMViewer that shows execution paths and the states of local variables in all the executions of a method. Using the tool, developers can investigate the behavior of a method by comparing execution paths. We expect that such a comparison is effective to understand test cases for a method. We would like to evaluate how the tool is used in software maintenance tasks in the future work.

Recovering the state of objects used in a method is also our future work. Although REMViewer uses actual field values accessed in a method, such a partial state of an object may be insufficient to understand what the accessed object is.

Finally, we would like to improve automatic classification of execution paths so that developers can easily identify normal cases and special cases of the behavior of a method in visualized paths.

6. ACKNOWLEDGMENTS

This work was supported by KAKENHI Nos.23680001 and 25220003.

7. REFERENCES

[1] D. Abramson, C. Chu, D. Kurniawan, and A. Searle. Relative debugging in an integrated development environment. *Software Practice and Experience*, 39(14):1157–1183, 2009.

[2] J. Bell, N. Sarda, and G. Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *Proc. of ICSE*, pages 362–371, 2013.

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. of OOPSLA*, pages 169–190, 2006.

[4] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[5] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proc. of ICSE*, pages 467–477, 2002.

[6] S. Joshi and A. Orso. SCARPE: A technique and tool for selective capture and replay of program executions. In *Proc. of ICSM*, pages 234–243, 2007.

[7] A. Ko and B. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proc. of ICSE*, pages 301–310, 2008.

[8] B. Lewis. Debugging backwards in time. In *Proc. of AADEBUG*, 2003.

[9] J. Ressia, A. Bergel, and O. Nierstrasz. Object-centric debugging. In *Proc. of ICSE*, pages 485–495, 2012.

[10] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *Proc. of ICSE*, pages 255–265, 2012.

[11] P. Sagdeo, V. Athavale, S. Kowshik, and S. Vasudevan. Precis: Inferring invariants using program path guided clustering. In *Proc. of ASE*, pages 532–535, 2011.

[12] J. Sillito, G. C. Murphy, and K. D. Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008.

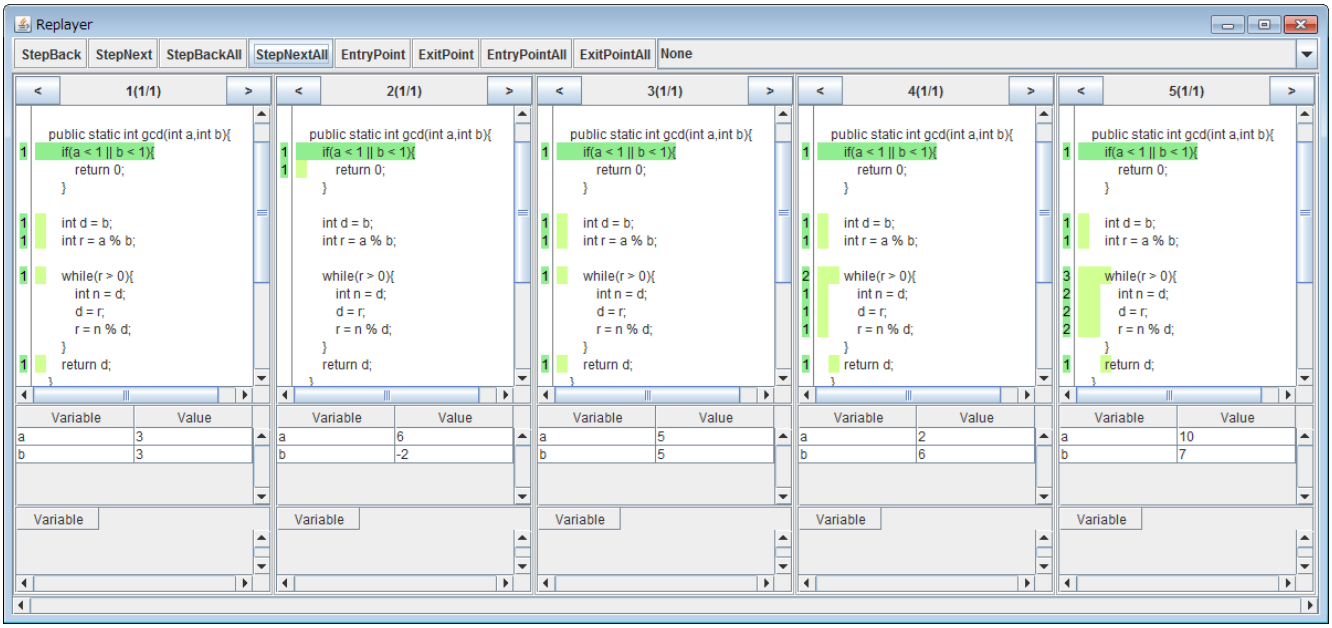

Figure 6: Five path views point to the first instruction.

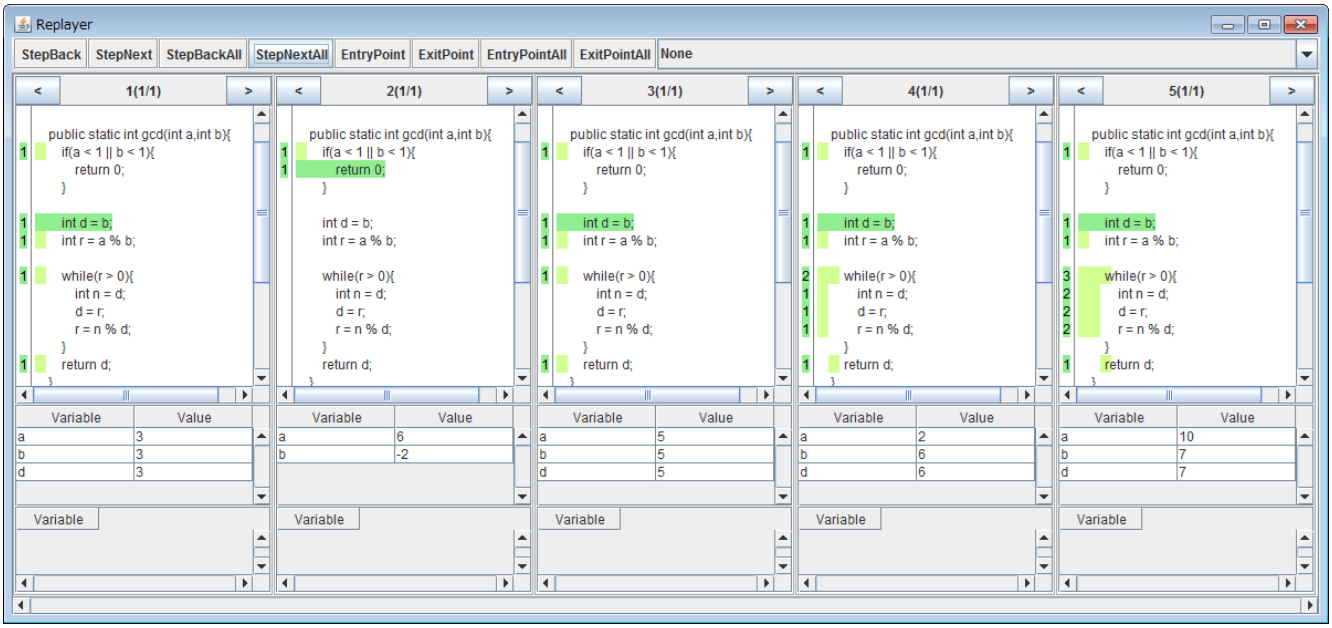

Figure 7: Each path view points to the second instruction in its execution path.

APPENDIX

A. SYNCHRONOUS SINGLE STEP EXECUTION

Synchronous single step execution is a basic feature of the tool. Figure 6 is a screenshot that shows five views that point to the first `if` instruction of the visualized method. Figure 7 is a screenshot after the synchronous single step execution. All the views point to the second instructions of the executions. A user can see that only one out of five paths entered into an `if` statement. A walk through of a method using the feature enables a user to easily compare when and how different paths are taken in each of executions.

A user can move the cursor in a path view to an arbitrary line by clicking on the line. Even if a method is too large for a walk through, a user may compare forward and backward execution paths from a line of interest in the method.