Visual Tracing for the Eclipse Java Debugger

Bilal Alsallakh, Peter Bodesinsky[1], Alexander Gruber[2], Silvia Miksch
Centre of Visual Analytics Science and Technology (CVAST)
Institute of Software Technology and Interactive Systems
Vienna University of Technology
Vienna, Austria
Email: {alsallakh, miksch}@cvast.tuwien.ac.at, {[1]e0304343, [2]e0625633}@student.tuwien.ac.at

***Abstract*—In contrast to stepping, tracing is a debugging technique that does not suspend the execution. This technique is more suitable for debugging programs whose correctness is compromised by the suspension of execution. In this work we present a tool for visually tracing Java programs in Eclipse. Tracepoint hits are collected on a per-instance basis. This enables finding out which tracepoints were hit for which objects at which time. The interactive visualization provides detailed information about the hits such as thread, stack trace, and assigned values. We implemented the tool as an Eclipse plugin that integrates with other features of Eclipse Java debugger. In an informal evaluation, developers appreciated the utility of our method as a solution in the middle between full tracing and stop-and-go debugging. They suggested scenarios in which our tool can help them in debugging and understanding their programs.**

I. Introduction

Eclipse has emerged as a mainstream IDE for the Java programming language. The Eclipse Java debugger allows the developer to perform program animation (also referred to as stepping) by setting several types of breakpoints. Breakpoints enable pausing the execution when a specific location in the code is reached, when a specific condition is met, or when a specific variable is being accessed. In some applications, it is not feasible for the debugger to interrupt the program's execution long enough for the developer to learn anything helpful about its behavior [1]. If the program's correctness depends on its real-time behavior, delays introduced by the debugger might cause the program to change its behavior, or perhaps fail, even when the code itself is correct. Examples of that arise when debugging application interfaces that involve user interaction. When a breakpoint defined in a method or on a variable that is involved in painting is hit, the program stops to respond to user interaction. Moreover, once the execution is resumed, the GUI needs to be repainted, which causes the breakpoint to be hit again. If the first hit does not reveal the bug, it would be difficult for the developer to reach the state that would reveal it.

Two features of the Eclipse debugger can be used to avoid unwanted suspension of execution. One feature is to add conditions to breakpoints. Another feature is to set the hit count for a breakpoint: the breakpoint will stop the execution only if it has been (silently) hit a specific number of times. To choose the appropriate hit count or condition, the developer should have considerable knowledge about the logic behind the source-code being debugged.

Another way to avoid unwanted suspension is to avoid using breakpoints at first. Instead, the developer can manually insert code snippets at the desired locations in the source code. These snippets print out the values that need to be checked to the console. After the code is executed, the developer can check the console to spot unexpected values. This mimics a debugging aid offered by some debuggers called tracepoints. In this work we propose a visual method for tracing Java programs in Eclipse. Our method fills a place in the middle between traditional stop-and-go debugging and full tracing.

II. Related Work

The use of tracing as debugging aid dates back to the 1970s with several debuggers offering the possibility to add tracepoints in programs [2]. When a tracepoint is hit during execution, information about the current program state as well as the values of user-defined expressions are printed out. Fig. 1 shows traces generated by GNU Debugger [1]. Table I lists keywords that can be used to specify the information in these traces. Tracepoints are useful in performing post-mortem debugging, i.e. debugging a program after it has already crashed. However, the textual output makes it cumbersome to relate the traces especially with multiple tracepoints.

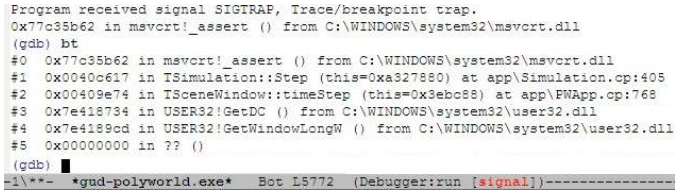

Fig. 1. A trace in GNU Debugger [1]. Each line shows available information about a tracepoint hit (such as source location and parameter values).

Keyword	Evaluates to
$ADDRESS	The address of the instruction
$CALLER	The name of the function that called this function
$CALLSTACK	The state of the call stack
$FUNCTION	The name of the current function
$PID	The ID of the process
$PNAME	The name of the process
$TID	The ID of the thread
$TNAME	The name of the thread

TABLE I
Tracepoint Keywords in GNU Debugger [3].

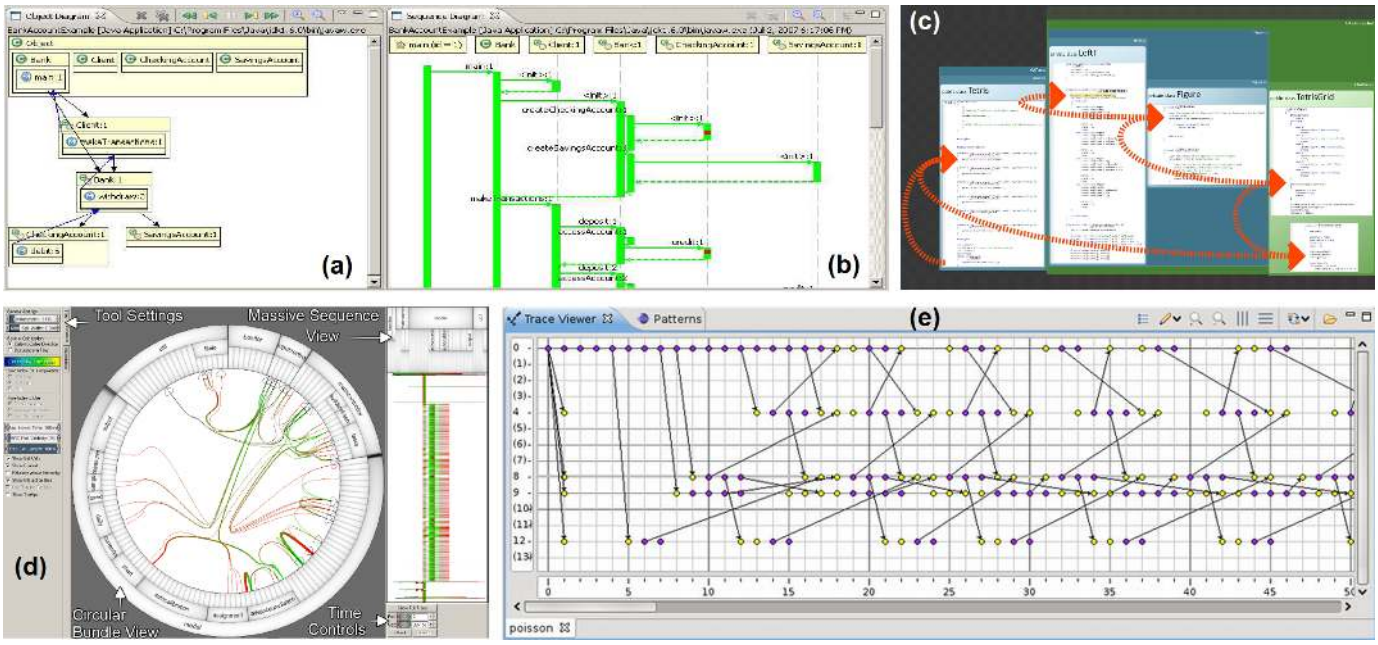

Fig. 2. Visualizing execution traces: (a, b) object and sequence diagrams in the JIVE plugin for Eclipse [4], (c) Code Canvas [5]: the arrows represent the call sequences, (d) linking a circular view of relations with a view of massive sequences [6], (e) the event graph of MPI communication traces [7].

Several approaches have been proposed for visualizing execution traces. JIVE [4] is an environment for visual debugging of Java programs. The current state of execution is depicted through an enhanced object diagram (Fig. 2a). The history of execution is depicted by a sequence diagram (Fig. 2b). These diagrams can be used to formulate queries over the program execution history and the runtime states. This constitutes a declarative approach to debugging. The JIVE plugin for Eclipse offers tight integration with other functionalities of the Eclipse IDE. However, JIVE might not be applicable for debugging large programs, both due to the overhead of collecting the runtime information, and due to visual limits.

Code Canvas [5] is a zoomable surface for software development in Visual Studio. A canvas houses editable forms of project documents. It also allows multiple layers of visualization over those documents. One layer can be used to visualize the stack trace at the runtime by means of arrows (Fig. 2c). It offers no view that shows trace information over time.

Cornelissen et al. [6] proposed linked views to visualize execution traces (Fig. 2d). A circular view shows which components of the program call each other. A massive sequence shows single calls in details. While this approach is highly scalable, it does not show variable values. It is more suited for program understanding than for data-driven debugging.

Klausecker et al. [7] proposed a method for handling large event traces using the Trace Viewer plugin of g-Eclipse (Fig. 2e). They used a pattern matching technique to simplify the debugging of large message passing parallel programs.

The method presented in this work uses simple and efficient metaphors for defining tracepoint and visualizing trace information to aid debugging Java programs in Eclipse.

III. From Breakpoints to Tracepoints

The Eclipse Java debugger is built upon the API of Java Debug Interface (JDI), which is part of the Java Development Toolkit. This API enables adding requests to monitor JVM events such as `BreakpointEvent`. When an event occurs, the debugger gets a notification and the thread in which this event took place can be obtained. For each frame in the stack trace of this thread the following information can be obtained:

- The source Java file in which the execution at this frame has taken place (or null if the source is not available).
- The method and line number (if available).
- The **`this`** object or **`null`** if the method is static.

The Eclipse debugger uses this information when a breakpoint is hit. It shows the stack trace for the suspended thread in the ”Debug” view. For the selected frame in this trace, Eclipse highlights the corresponding line number in its source file, and displays the **`this`** variable in the ”Variables” view.

To make a breakpoint behave like a tracepoint, our plugin disables it. This causes Eclipse debugger to ignore the `BreakpointEvent`s, but allows the plugin to track the above information. Also, a timestamp for each such silent hit is recorded. This is computed as the number of elapsed milliseconds since the program was started. In an object-oriented context, it is meaningful to divide the silent hits into groups according to the object they belong to. For line breakpoints and method breakpoints, the object is the **`this`** instance on which the corresponding method is executed. For watchpoints, the object is the instance to which the field variable being watched belongs. We call the hits in each group, the breakpoint history for the respective instance. For a breakpoint in a static method or on a static field, only one history is tracked.

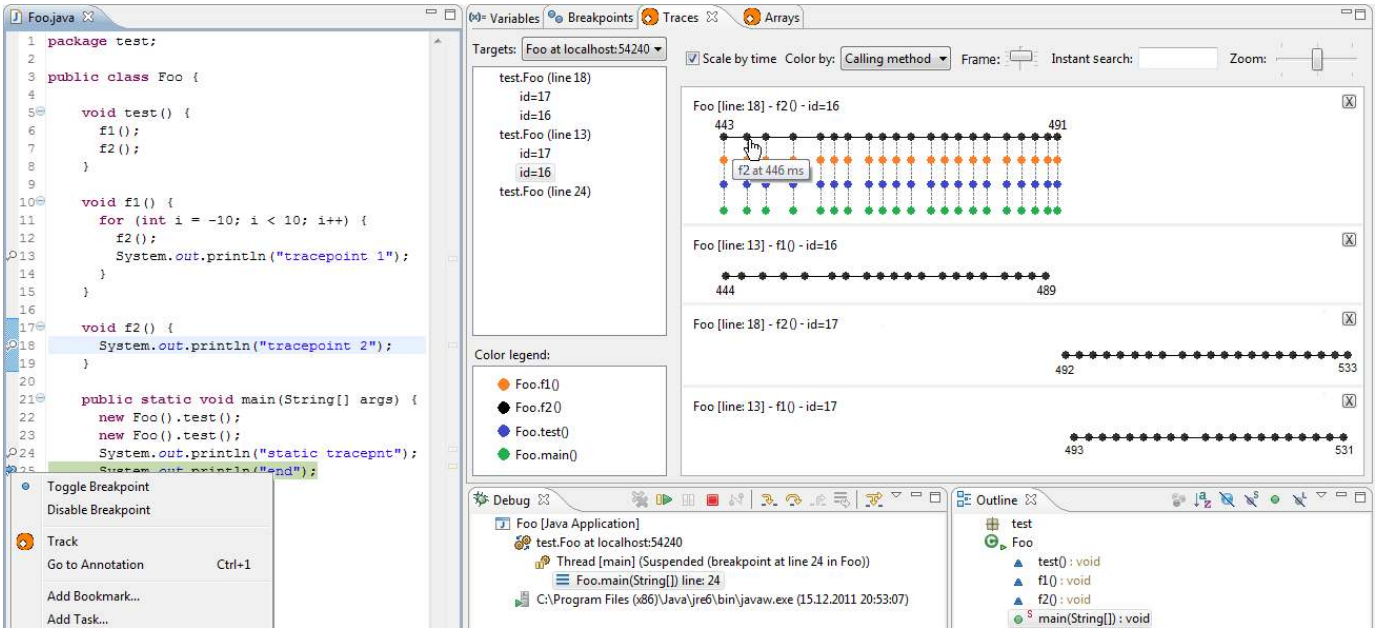

Fig. 3. The "Traces" view of our Eclipse plugin. Four traces resulting from the listing to the left are visualized. Scaling by time is activated to reveal the temporal relationship between the histories. A dot representing a hit is clicked and the corresponding location in the code is highlighted in light blue.

IV. Trace Visualization

A breakpoint history is basically visualized as a line chart. The dots on the horizontal line represent the single hits in this history. The hits are ordered in their chronological order. Optionally, the stack trace for each hit can be visualized on a vertical dashed line. The frames of this trace are represented by dots on this line in their stack order from top to bottom (Fig. 3). The user can select the maximum stack depth to be visualized. A similar metaphor is used in Saturn and Shark profilers for Mac OS X. The dots in the histories can be placed at a fixed interval or at locations scaled by their timestamps. The former case is simpler and reduces overlapping in case of a highly uneven distribution of dots over time. The latter case is useful for understanding this distribution or for comparing multiple histories over time. Horizontal zooming and panning help in focusing on a specific time range. By hovering the mouse over a dot more details about the hit it represents are shown in a tooltip, such as the timestamp and method name. When a dot is double-clicked, the corresponding line-of-code is highlighted in its source file in the Java editor. The dots can be colored by the calling methods at a given stack depth (Fig. 3) or by the calling threads (Fig. 4).

Fig. 3 shows the implemented "Traces" view in Eclipse. The tracing-enabled breakpoints are shown in the left panel in this view. Under each breakpoint, instances of its class that have recorded histories are listed. These instances can be inspected on the fly in the Eclipse variable inspection popup. In case of a line breakpoint in a static method or a watchpoint on a static field, the list entry represents its unique history. When a history is double clicked, its visualization is added to the right panel. The visualization is updated upon a new hit.

V. Variable History

Tracing can also be enabled for a watchpoint on a field (also referred to as a data breakpoint). In this case, the history associated with a specific instance records the values assigned to the field's variable in this instance. The user can decide to show all hits or only hits that represent write accesses. In the former case, different visual representations for read hits and write hits are used. We distinguish between two cases for the visualization depending on the field type:

1) Numerical Primitive Types: Except for `boolean` and `char`, the values in the history of a primitives type can be naturally visualized in a 2-dimensional step chart. The x-axis represents the time or the order of the hits. The y-axis represents the numerical value. In Fig. 4-(c) the dots representing numerical values are colored by the threads which assigned them. One can notice the race condition caused by an unsynchronized critical area.

2) Reference Types / `char`: Reference types are subclasses of `Object`. Values of this type are visualized as dots on a line chart. Additional functionalities help in analyzing these values. Hovering a dot will show a tooltip of the textual (`toString()`) representation of its value. An instant search highlights dots whose textual representations contain a specific text (Fig. 4-b). Other values are blended. This helps in determining if specific values were assigned to the watched field variable. Right-clicking on a dot opens the Eclipse variable inspection popup to show its value in details. To determine the runtime types of the values, the dots can be colored according to the different types. By default, null values in the history are visualized as gray dots (Fig. 4-b). This helps in quickly finding potential sources of errors caused by null values.

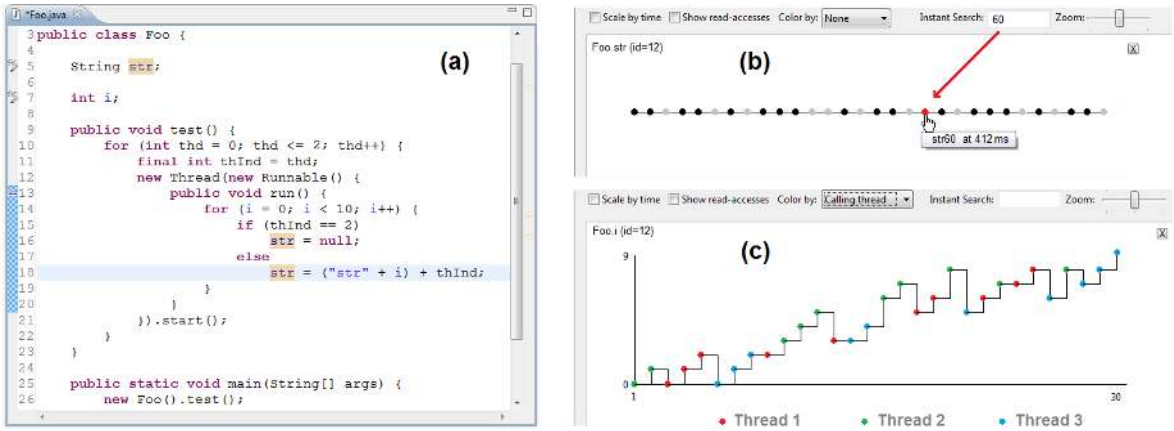

Fig. 4. Variable histories: (a) a sample Java program, (b) the history of a variable of reference type. Gray dots represent null values. An instant search is applied and the search result is highlighted in red, (c) the history of a numerical variable. Dots are colored by the thread which assigned their values.

VI. Discussion

We performed an informal evaluation of our tool with ten practitioners that have been using Eclipse for several years. We used small Java programs that intend to illustrate the visualization and interaction metaphors. No tasks were required to be solved. We were mainly interested in finding out how intuitive and familiar our method for visual tracing is. The developers found the metaphors intuitive and easy to interpret. The ability to define tracepoints out of existing Eclipse breakpoints made it easy to understand the tracing process. For the visualization of the history of a reference variable, one developer preferred the use of a tabular view instead of a line chart. Another request was to use hierarchical abstraction instead of the matrix-like visualization for stack traces. When asked if the visualizations can help them solving real-world problems, two developers mentioned scenarios related to GUI debugging. Another developer mentioned understanding the runtime behavior of algorithms. Debugging multi-threaded applications was also mentioned as an application scenario.

The presented method is designed to visualize relatively short traces (with number of hits ≤ 100), and a few number of traces (≤ 7) simultaneously. This is sufficient for many debugging tasks that arise in practice, even in large programs. Spatial subsampling [6] and dot aggregation techniques, as well as nonlinear transformation of the time axis [8] need to be investigated to handle a large number of hits. The current implementation registers its own `EventRequest` in the JVM. This might cause deadlocks in multi-threaded programs. This can be avoided by intercepting the listeners already registered by Eclipse debugger (this might require a change in Eclipse).

VII. Conclusion

We have presented a novel tool for tracing Java programs in Eclipse. Our tool enables altering conventional Eclipse breakpoints to tracepoints and collecting runtime information from them. The hits for each breakpoint are collected on a per-instance basis which is well-suited for an object-oriented language. Using interactive visualization, these hits can be explored and related to the corresponding location in the source-code. The visualization helps also in examining time and thread information, as well as values and types for data breakpoints. Compared with existing Eclipse plugins for visual debugging, our plugin offers a simpler and more familiar process for defining and visualizing the data to be traced. Informal evaluation showed that the per-instance histories and the metaphor of line charts are easy to understand. Some developers suggested scenarios in which visual tracing can help them in debugging and understanding their programs.

Future work should focus on better integration with Eclipse to acquire the data without registering additional listeners in the JVM. Also, it should address scalability issues as well as features requested by the test subjects. Additionally, more formal evaluation needs to be performed. We are currently preparing programs that contain deliberate bugs. The developers will be asked to find the bugs with and without our tool.

Acknowledgement This work was supported by CVAST Centre for Visual Analytics Science and Technology (initiative #822746).

References

[1] R. M. Stallman, R. Pesch, and S. Shebs, *Debugging with GDB - The GNU Source-Level Debugger*. GNU Press, Mar. 2002.

[2] G. M. Bull, "Dynamic debugging in BASIC," *Comput. J.*, vol. 15, no. 1, pp. 21–24, 1972.

[3] S. Akhter and J. Roberts, "Multi-threaded debugging techniques," in *Multi-Core Programming*. Intel Press, 2006, pp. 215–236.

[4] J. K. Czyz and B. Jayaraman, "Declarative and visual debugging in eclipse," in *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*. New York, NY, USA: ACM, 2007, pp. 31–35.

[5] R. DeLine and K. Rowan, "Code Canvas: Zooming towards better development environments," in *Proc. of the 32nd International Conference on Software Engineering*, vol. 2, 2010, pp. 207–210.

[6] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk, "Execution trace analysis through massive sequence and circular bundle views," *J. Syst. Softw.*, vol. 81, pp. 2252–2268, 2008.

[7] D. K. Thomas Köckerbauer, Christof Klausecker, "Scalable parallel debugging with g-Eclipse," in *Tools for High Performance Computing 2009: Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing*. Springer, 2009, pp. 215–236.

[8] J. Trümper, J. Bohnet, and J. Döllner, "Understanding complex multi-threaded software systems by using trace visualization," in *Proc. of the international symposium on software visualization*, 2010, pp. 133–142.