Visual Analytics for Concurrent Java Executions*

Cyrille Artho
KTH Royal Institute of Technology
Email: artho@kth.se

Monali Pande
KTH Royal Institute of Technology
Email: monalip@kth.se

Qiyi Tang
Imperial College London
Email: qiyi.tang71@gmail.com

***Abstract*—Analyzing executions of concurrent software is very difficult. Even if a trace is available, such traces are very hard to read and interpret. A textual trace contains a lot of data, most of which is not relevant to the issue at hand. Past visualization attempts either do not show concurrent behavior, or result in a view that is overwhelming for the user.**

We provide a visual analytics tool, VA4JVM, for error traces produced by either the Java Virtual Machine, or by Java Pathfinder. Its key features are a layout that spatially associates events with threads, a zoom function, and the ability to filter event data in various ways. We show in examples how filtering and zooming in can highlight a problem without having to read lengthy textual data.

***Index Terms*—Execution trace visualization, visual analytics**

I. Introduction

Program executions are complex, and the visualization of execution traces can be a key towards understanding them, and finding the root cause of problems. Tools that derive sequence diagrams [1] from executions, or visualize message passing [2] are not ideal for thread-level concurrency, because different threads are not clearly distinct in the way the trace is shown.

In our work, program traces may originate from a concurrent program running under the Java Virtual Machine (JVM), or on the dedicated concurrency analysis tool Java Pathfinder (JPF) [3], [4]. On the JVM, a recorded trace is an approximation of the executed program behavior, because event recording in an unrestricted concurrent execution environment is not atomic [5]. On JPF, the trace is a precise representation of the failure. Both cases have in common that traces are long and complex, and include a lot of irrelevant data, such as long event sequences that merely set up the program.

On the JVM, we record a trace until the program terminates due to an assertion failure or uncaught exception. JPF checks a given program against uncaught exceptions, deadlocks, and assertion violations. Without visualization, trace data in textual format has an overwhelming amount of detail. For any non-trivial error, traces exceed thousands of lines of text.

Figure 1 shows a partial trace of the Dining Philosophers [6] example, generated by JPF. As can be seen, the trace is cluttered with details such as parsing command line parameters. The first twelve transitions merely initialize all threads; the actual part that leads to a deadlock begins with transition 13. The only clue to a user that transition 13 is the first "interesting" transition is that the ID of the active thread changes from 0 to 1, something that is easily overlooked when analyzing long traces.

* Part of this work has been supported by Google Summer of Code.

```
==================================================== trace #1
------------------------------------ transition #0 thread: 0
gov.nasa.jpf.vm.choice.ThreadChoiceFromSet {id:"ROOT",1/1,...}
      [3157 insn w/o sources]
  DiningPhil.java:46: static int nPhilosophers = 6;
  DiningPhil.java:1 : /*
      [1 insn w/o sources]
  DiningPhil.java:49: if (args.length > 0){
  DiningPhil.java:50: nPhilosophers = Integer.parseInt(args[0]);
      [2 insn w/o sources]
  DiningPhil.java:50: nPhilosophers = Integer.parseInt(args[0]);
...
  DiningPhil.java:59: Philosopher p = new Philosopher(...);
  DiningPhil.java:60: p.start();
      [1 insn w/o sources]
------------------------------------ transition #1 thread: 0
gov.nasa.jpf.vm.choice.ThreadChoiceFromSet {id:"START",1/2,...}
      [2 insn w/o sources]
  DiningPhil.java:58: for (int i = 0; i < nPhilosophers; i++) {
  DiningPhil.java:59: Philosopher p = new Philosopher(...);
  DiningPhil.java:29: public Philosopher(Fork left, Fork right) {
      [27 insn w/o sources]
------------------------------------ transition #2 thread: 0
gov.nasa.jpf.vm.choice.ThreadChoiceFromSet {id:"LOCK",1/2,...}
      [119 insn w/o sources]
  DiningPhil.java:30: this.left = left;
  DiningPhil.java:31: this.right = right;
...
------------------------------------ transition #13 thread: 1
gov.nasa.jpf.vm.choice.ThreadChoiceFromSet{id:"TERMINATE",1/5,...}
...
```

Fig. 1. Excerpt of an error trace in JPF (Dining Philosophers). Ellipses represent elided parts.

We present a pragmatic visualization that adopts a tabular layout to show the threads spatially, while keeping the option to show the original (detailed) transition information in full text. We furthermore have a facility that selectively highlights and filters out operations that are expected to contribute to the failure, allowing the user to mine a trace interactively and understand its meaning using visual analytics [7].

The intended audience consists of developers who want to analyze concurrent execution traces, be this for software validation or root cause analysis of a failed software execution.

II. Visualization of JPF traces

When used with Java Pathfinder (JPF), our visual analytics tool [8] is implemented on top of the JPF shell [9]. It has a navigation panel on the left, which allows a user to expand/collapse all transitions at once, or to highlight individual aspects of all transitions. Operations that can be highlighted include `wait`/`notify`, starting new threads and joining a thread, lock usage, field accesses, and method calls. The latter two options are accessed by a drop-down list (see Figure 2). Highlighted parts of the source code are shown in up to 15 distinct colors, both in the expanded and collapsed modes.

The trace is split into transitions. Each transition (in JPF) represents a sequence of thread-local actions. The user starts

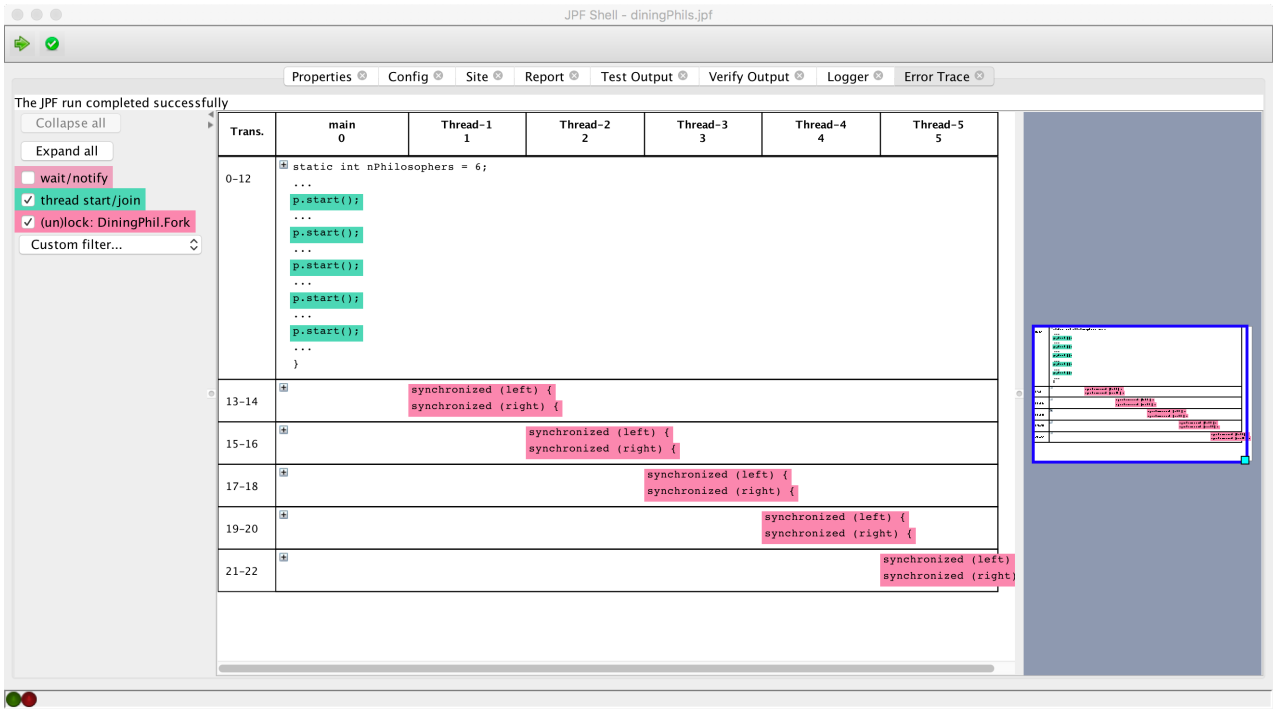

Fig. 2. Screenshot of the application. Thread starts and lock acquisitions are highlighted. The deadlock between the threads is immediately visible. The collapsed summary hides the command line parsing that reduces the number of threads from the initial value of six to five.

with a fully collapsed summary, which shows only the first and last line of code for each transition. On the right side, a "world map" (overview) is shown. This view can simplify the navigation in large traces and also supply a condensed representation of key thread attributes [2].

VA4JVM combines a bottom-up analysis of transitions with top-down filtering for key events. Typical selections for highlighting include "(un)lock" for debugging traces highlighting a data race or deadlock, and field accesses or method calls for more complex problems such as NullPointerExceptions. With each selection, the central panel is automatically updated; the summarized error trace is extended if necessary, so the selected attributes are always shown in the summarized view as well.

As the tool is designed to analyze concurrent executions, the special method calls `start`, `join`, `wait`, and `notify` (including `notifyAll`) are given as preset filters. Filters will automatically highlight the chosen event, e. g., an access to a field, even in collapsed transitions. Preliminary experiments have shown that compared to a textual log, VA4JVM greatly cuts down the amount of data that is presented to the user, and allows a step-wise analysis of a trace when the root cause is suspected (by filtering) and even when it is still unknown (by zooming in on individual transitions from the bottom up) [10]. These experiments include examples with several threads exhibiting data races, deadlocks, and examples that use reader-writer locks or producer-consumer patterns [10].

III. Adaptation of JPF-visual for the JVM

While Java Pathfinder is, thanks to its exhaustive analysis of thread schedules, a very powerful tool to detect concurrency problems, the state space explosion limits software model checking from being applied to larger programs. However, there exist many approaches to find concurrency issues based on run-time monitoring, such as data race detection [11], [12].

These algorithms have in common that they show a partial snapshot of the state in which a (potential) data race is found, but not the execution history that leads to that state. Furthermore, their output is purely textual. VA4JVM covers these gaps: It records the execution history up to the error, and displays the result graphically. It also supports a detailed analysis of all field accesses of a certain type, in order to follow the history of field accesses up to the data races.

We have implemented VA4JVM by extending the visualization of JPF traces [8] and adding the capability to record and present events from executions on the standard Java Virtual Machine (JVM) [13], [10]. We record execution events with an extended version of AspectJ [14] that covers the occurrence `synchronized` blocks in Java as well as events relating to method calls and field accesses [15]. Our instrumentation-based event recording may not always capture events in the order in which they take effect on the program, as this is impossible without changing the JVM due to the Java memory model [16]. We also treat calls to key functions of class `ReentrantLock` specially, so a user can analyze locking with that utility class in a uniform way with standard locking using `synchronized` blocks.

Our instrumented code uses adapters [17] to represent data from the JVM and from JPF with a unified interface. As our back-end partially relies on instruction-level events, we create

Fig. 3. Screenshot of VA4JVM, for analyzing a reader-writer example with one reader and one writer thread. The left panel shows the filters, the central panel part of the execution (zoomed out), and the right panel shows the thread state view. In the thread state view, the yellow (green) bars represent locking (unlocking) operations; a vertical line shows the lifetime of a thread while holding a lock (in yellow) or no lock (in green).

synthetic events to represent method calls recorded by AspectJ the corresponding bytecode instruction.

VA4JVM has the same analysis capabilities for event traces from the JVM, as it has for JPF traces (see Fig. 3). The only exception is the occurrence of a deadlock, a situation where no thread can continue. Because our visualization occurs after a program terminates, and depends on observing the termination of the last active thread, we have to detect deadlocks at runtime at the point just before the last unsuccessful (deadlocking) operation takes place. We currently implement this for special cases [10]; a generalization is future work.

In our example of VA4JVM for JVM traces, Fig. 3 shows the same left and central panels as in Fig. 2, but a *thread state view* in the right panel. This view, available as an alternative to the "world map", shows the lifetime of each thread (as a vertical bar) and its state (green = runnable; yellow = holding at least one lock; red = blocked). We can see that lock synchronization is heavily used by the program, due to the implementation of reader/writer locks that uses low-level locks internally. The reader thread terminates early in this example, as illustrated by the end of the vertical bar that represents the thread state over its lifetime.

IV. Related Work

We compare our work with visual dashboards that are common for log analytics and DevOps, and execution trace analytics tools that are popular for embedded software.

A. Log data analysis

Log analysis has been used to find integration problems, and to localize the root cause of failures [18]. System and integration testing have become log-driven activities [19], because log analysis is appropriate for large, distributed, and heterogeneous platforms [19].

DevOps relies on monitoring tools that typically provide dashboard-like overviews and detailed views of individual logs or data points computed from it [20]. Some tools also provide trace analysis capabilities, at the level of monitoring functions but not the execution between functions (and without thread-specific visualization). These include OverOps [21], which provides data tracing and analysis; OpenZipkin [22], which features monitoring and filtering; and Shiviz [23], which provides a small set of log analytics functions.

B. Visualization of execution traces

Platforms for visualizing traces from embedded software on the Linux kernel are currently the most advanced and accessible ones. They include a tool to visualize message passing [2], LTTng (trace analysis of Linux kernel events) [24], TraceCompass (for a visual analysis of LTTng traces) [25], and the APP4MC component in Amalthea, which analyzes multi-process programs [26]. These platforms have fixed visualizations that support interval-like views of events and system states that are obtained from system events. They currently do not visualize thread-based concurrency.

To visualize test executions at a higher level, Dine et al. [27] represent TTCN-3 (Testing and Test Control Notation) test case executions graphically. ASTRO [28] extracts information from test execution log files and represents it through three visual metaphors or perspectives: an overview (a general visualization of the test case execution), callers and callees (classes and methods that were called before and after a selected method execution), and a history perspective (chronological test executions). Jones et al. [29] provide a visual mapping of the participation of each program statement in a test suite. A similar approach is developed by Gouveia et al. [30], where three different interactive visualizations are used to navigate through the project structure.

Recent work for the Java platform includes a tool that shows a tabular overview of program counters and memory contents of test executions [31], and SDExplorer [1], a very scalable tool that generates UML sequence diagrams. Like our tool, these tools are integrate event collection with visualization; however, they do not feature concurrency-related views that allow the user to visually distinguish threads.

V. Conclusion and Future Work

Our visual analytics tool VA4JVM records error traces and displays them graphically in a way that multiple threads are clearly separated. VA4JVM allows a user to zoom in, filter, and highlight parts of concurrent execution traces. This makes it possible to look at parts of the trace in higher detail, and provides an interactive, dynamic visualization.

We plan to refine and extend the visualization in various ways. The current version of field/method call highlighting is limited to static information from the available data in the trace. A more detailed highlighting could distinguish different object instances (rather than just types). This will produce more trace data, which will then require a dedicated back-end database for storage and analysis.

Other improvements include more complex filters and the ability to hide threads to reduce visual clutter. Furthermore, we want to add more "zoom levels" for transitions, such as a low-level view with individual bytecode instructions [31], and alternative views such as a diagrammatic visualization [32]. The "world map" will also be enriched with some key information on threads, such as their state.

References

[1] K. Lyu, K. Noda, and T. Kobayashia, "SDExplorer: A generic toolkit for smoothly exploring massive-scale sequence diagram," in *Proc. 26th Conf. on Program Comprehension*, ser. ICPC 2018. New York, NY, USA: ACM, 2018, pp. 380–384.

[2] D. Kvarfordt and E. Bogren, "Visualization of message passing in an embedded system," Master's thesis, Chalmers University of Technology and University of Gothenburg, 2014.

[3] C. Artho and W. Visser, "Java Pathfinder at SV-COMP 2019 (competition contribution)," in *TACAS (3)*, ser. Lecture Notes in Computer Science, vol. 11429. Springer, 2019, pp. 224–228.

[4] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated Software Engineering Journal*, vol. 10, no. 2, pp. 203–232, 2003.

[5] J.-D. Choi and H. Srinivasan, "Deterministic replay of Java multithreaded applications," in *Symposium on Parallel and Distributed Tools: Proceedings of the SIGMETRICS Symposium on Parallel and distributed tools*, vol. 3, no. 04, 1998, pp. 48–59.

[6] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[7] K. A. Cook and J. J. Thomas, *Illuminating the Path: The Research and Development Agenda for Visual Analytics*. Los Alamitos, CA, United States(US).: IEEE Computer Society, 2005.

[8] Q. Tang and C. Artho, "jpf-visual," https://github.com/qiyitang71/jpf-visual, 2019, last accessed: 2019-08-23.

[9] S. Badame, P. Mehlitz, and P. Hudececk, "jpf-shell," https://jpf.byu.edu/hg/jpf-shell, 2016, last accessed: 2019-05-30.

[10] M. Pande, "Visual analytics tool for Java virtual machine execution traces," Master's thesis, KTH Royal Institute of Technology, 2019.

[11] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Computer Systems*, vol. 15, no. 4, pp. 391–411, 1997.

[12] C. Artho, K. Havelund, and A. Biere, "High-Level Data Races," in *VVEIS 2003: The First Int. Workshop on Verification and Validation of Enterprise Information Systems*. Angers, France: ICEIS Press, 2003.

[13] M. Pande, "VA4JVM," https://github.com/monalip/VA4JVM, 2019, last accessed: 2019-06-04.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "An overview of AspectJ," *LNCS*, vol. 2072, pp. 327–355, 2001.

[15] E. Bodden and K. Havelund, "Aspect-oriented race detection in Java," *IEEE Trans. Softw. Eng.*, vol. 36, no. 4, pp. 509–527, 2010.

[16] J. Manson, W. Pugh, and S. V. Adve, *The Java memory model*. ACM, 2005, vol. 40, no. 1.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, ser. Addison-Wesley Professional Computing Series. New York, USA: Addison-Wesley Publishing Company, 1995.

[18] L. Mariani, F. Pastore, and M. Pezzè, "Dynamic analysis for diagnosing integration faults," *IEEE Trans. Softw. Eng.*, vol. 37, no. 4, pp. 486–508, 2011.

[19] A. J. Oliner, A. Ganapathi, and W. Xu, "Advances and challenges in log analysis," *Commun. ACM*, vol. 55, no. 2, pp. 55–61, 2012.

[20] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "DevOps," *IEEE Software*, vol. 33, no. 3, pp. 94–100, 2016.

[21] OverOps, "The OverOps AIOps Platform," https://www.overops.com/, 2019, last accessed: 2019-06-04.

[22] Zipkin, "OpenZipkin: A distributed tracing system," https://zipkin.io/, 2019, last accessed: 2019-06-04.

[23] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst, "Debugging distributed systems," *Commun. ACM*, vol. 59, no. 8, pp. 32–37, 2016.

[24] M. Desnoyers and M. Dagenais, "The LTTng tracer : A low impact performance and behavior monitor for GNU / Linux," in *OLS (Ottawa Linux Symposium)*, 2006, pp. 209–224.

[25] The Eclipse Foundation, "TraceCompass," https://www.eclipse.org/tracecompass/, 2019, last accessed: 2019-06-04.

[26] R. Hoettger, H. Mackamul, A. Sailer, J. Steghöfer, and J. Tessmer, "APP4MC: application platform project for multi- and many-core systems," *it - Information Technology*, vol. 59, no. 5, pp. 243–251, 2017.

[27] G. Din, J. Zander, and S. Pietsch, "Test execution logging and visualisation techniques," in *17th Int. Conf. Software and Systems Engineering and their Applications, Paris, France*, 2004.

[28] D. Castro and M. Schots, "Analysis of test log information through interactive visualizations," in *Proc. 26th Conf. on Program Comprehension*, ser. ICPC 2018. New York, NY, USA: ACM, 2018, pp. 156–166.

[29] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proc. 24th Int. Conf. on Software Engineering*, ser. ICSE 2002. New York, NY, USA: ACM, 2002, pp. 467–477.

[30] C. Gouveia, J. Campos, and R. Abreu, "Using HTML5 visualizations in software fault localization," 09 2013, pp. 1–10.

[31] S. Wesonga, E. G. Mercer, and N. Rungta, "Guided test visualization: Making sense of errors in concurrent programs," in *26th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2011)*, 2011, pp. 624–627.

[32] C. Artho, K. Havelund, and S. Honiden, "Visualization of concurrent program executions," in *Proc. 2nd Int. Workshop on Software Architectures and Component Technologies (SACT 2007)*, vol. 2, Beijing, China, 2007, pp. 541–546.