

Back to the Future: Omniscient Debugging

Guillaume Pothier and Éric Tanter, *University of Chile*

A scalable debugger for Java integrated into Eclipse paves the way for practical omniscient debugging.

Debugging represents a major cost in the software development process. A 2002 US National Institute of Standards and Technology study established that software errors have an enormous cost on the US economy and mentioned that “software developers already spend approximately 80 percent of development costs on identifying and correcting defects.”¹ In an empirical study of debugging stories, Marc Eisenstadt found that the major reason why bugs are difficult to track down is the large temporal or spatial chasm between the root cause and the actual symptom of a bug;²

once a bug is precisely located, fixing it is often trivial. Unfortunately, most debuggers provide very limited assistance for temporal navigation, so programmers frequently have to resort to mental simulation of program execution.

Omniscient debuggers drastically improve the situation by enabling programmers to seamlessly navigate forward and backward in a buggy program’s execution history and easily find the root cause of errors through causal links.³ An omniscient debugger can thus have a high impact on the development process’s efficiency.

Omniscient debugging is far from a new idea: the first omniscient debugger, EXDAMS (Extendable Debugging and Monitoring System),⁴ dates back to 1969. While numerous systems have been proposed since then, omniscient debuggers still aren’t part of the typical development environment. Are the challenges of omniscient debugging a definitive barrier to its adoption?

This article presents TOD (trace-oriented debugger), a prototype scalable omniscient debugger for Java, which aims at making omniscient debugging practical, at last.

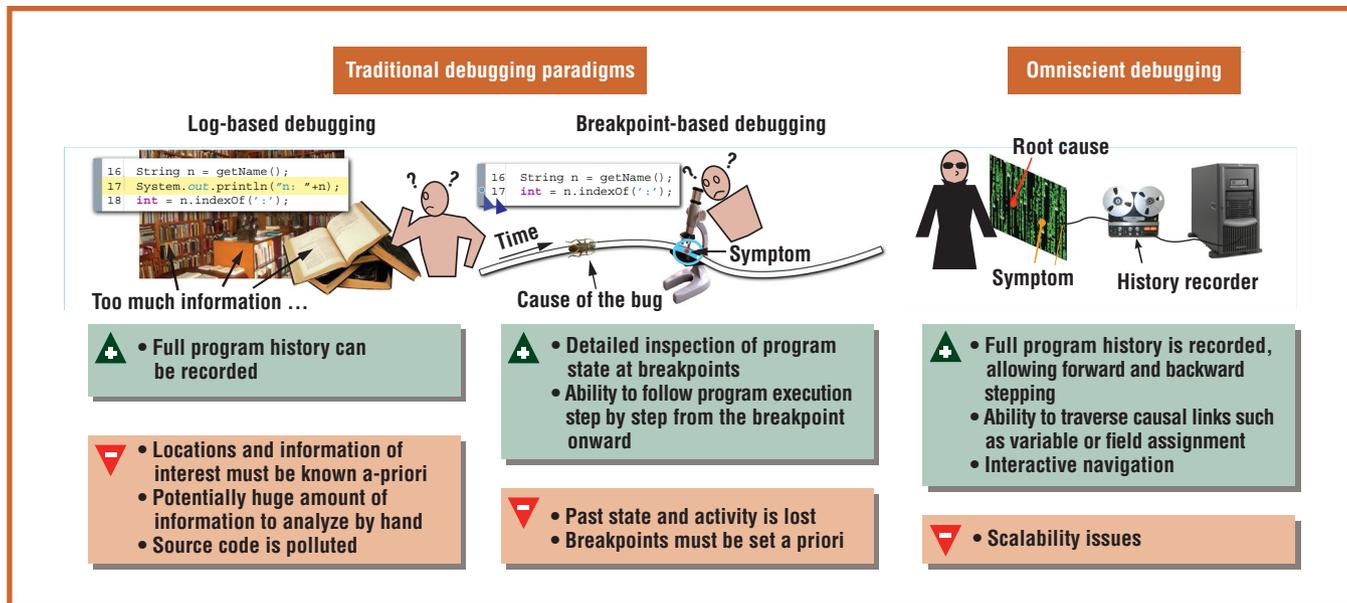
Omniscient Debugging in a Nutshell

So what is omniscient debugging about, and what challenges does it face? First, let’s briefly expose the traditional approaches to debugging and why they eventually fall short.

Traditional Approaches to Debugging

Figure 1 shows the two traditional approaches to debugging. Log-based debugging consists of inserting logging statements within the source code to produce an ad hoc trace during a program’s execution. This technique exposes the actual history of the execution but presents significant inconveniences: it requires cumbersome, widespread, and anticipated modifications to the source code, and it hardly scales when the programmer has to manually analyze traces.

Breakpoint-based debugging consists of running the program under a dedicated debugger tool, which lets the programmer pause the execution at determined breakpoints, inspect memory contents, and then continue execution step by step. Unfortunately, when the execution is paused,



information about the program's previous state and activity is limited to the current call stack. Developers using breakpoint-based debuggers are familiar with having to rerun the whole program many times with different sets of breakpoints to progressively home in on the bug.

Omniscient Debugging

Omniscient debuggers, also known as *back-in-time* or *reversible* debuggers, record the whole history, or *execution trace*, of a debugged program and let the user freely explore it. This approach combines the advantages of both log-based (past activity is never lost) and breakpoint-based debugging (interactive navigation, step-by-step execution, and complete stack inspection). Omniscient debuggers simulate step-by-step execution both forward and backward, avoiding having to rerun the whole program many times to pinpoint the bug's root cause. More importantly, they make it possible to navigate through the history of a program by following causal links, so questions that would otherwise require a significant effort can be answered instantly—for instance, “When was variable *x* assigned a null value?” or “What was the state of object *o* when it was passed as an argument to method *foo*?”

Challenges

Although omniscient debugging has clear advantages over traditional approaches, it's still considered mostly unrealistic because of the important scalability issues it raises:

- Capturing the execution trace should not

cause too high an overhead on the debugged application.

- Execution traces grow very quickly and thus require fast and scalable storage.
- Queries on a possibly huge trace should be processed fast enough for the debugging environment to be responsive to user interaction.
- However large the execution trace, the developer must be able to rapidly locate the points of interest and establish meaningful relations between execution points.

While fully addressing all these challenges is a non-trivial task, it is possible to mitigate these issues to a large extent. Our work on TOD illustrates how we achieve practical omniscient debugging of Java programs.

Overview of TOD

TOD is a trace-oriented debugger for Java integrated into the Eclipse development environment (see Figure 2):⁵

- Instrumentation (phase 1). When the Java Virtual Machine (JVM) is about to load a class, the agent sends its bytecode to the weaver, which inserts event generation code into the class and then sends it back to the JVM.
- Event emission (phase 2). As the program runs, the instrumented code generates events and sends them to the event database. The sequence of generated events constitutes the execution trace.
- Storage and indexing (phase 3). The highly specialized event database stores events at

Figure 1. Approaches to debugging. Omniscient debugging is an alternative to traditional techniques such as log-based and breakpoint-based debugging.

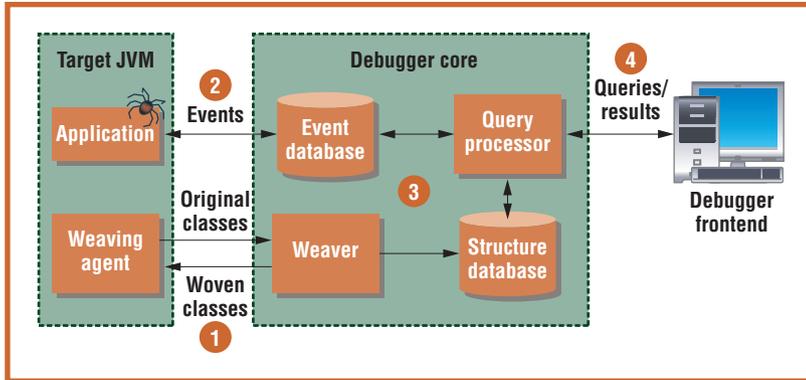


Figure 2. How TOD (trace-oriented debugger) works: architecture and operation. The four basic phases of an omniscient debugging session: (1) instrumentation, (2) event generation, (3) execution trace storage and indexing, and (4) interactive navigation.

a very high rate and indexes them to allow fast query processing. Additionally, a structure database stores static information about the debugged program, such as its classes and methods.

- Querying and navigation (phase 4). The developer navigates in the execution trace using the debugger front end, which is integrated into the Eclipse IDE.

By leveraging the very constrained nature of execution traces (events arrive almost in timestamp order and are never modified once emitted) and the fact that all omniscient debugging navigation actions can be computed using simple event filtering queries, we designed a very scalable system: the event database is parallelizable, and in our benchmarks on a 10-machine cluster, it handled a sustained input rate of 500,000 events per second and processed queries in fractions of a second on traces containing almost a billion events.⁵ However, trace capture causes a significant slowdown of the debugged program: up to 80 times in the worst case (a fully instrumented, CPU-intensive program), although it's possible to greatly reduce it by excluding parts of the program from instrumentation (for example, classes of the standard Java libraries).

From our experience, a single-machine setup is enough for relatively small execution traces (10 million events), and a two-machine setup (that is, one dedicated database machine in addition to the development machine) can comfortably handle traces of roughly 150 million events, which is enough to debug the event database itself, for instance. For larger traces, organizations that can afford it would benefit from a more powerful setup.

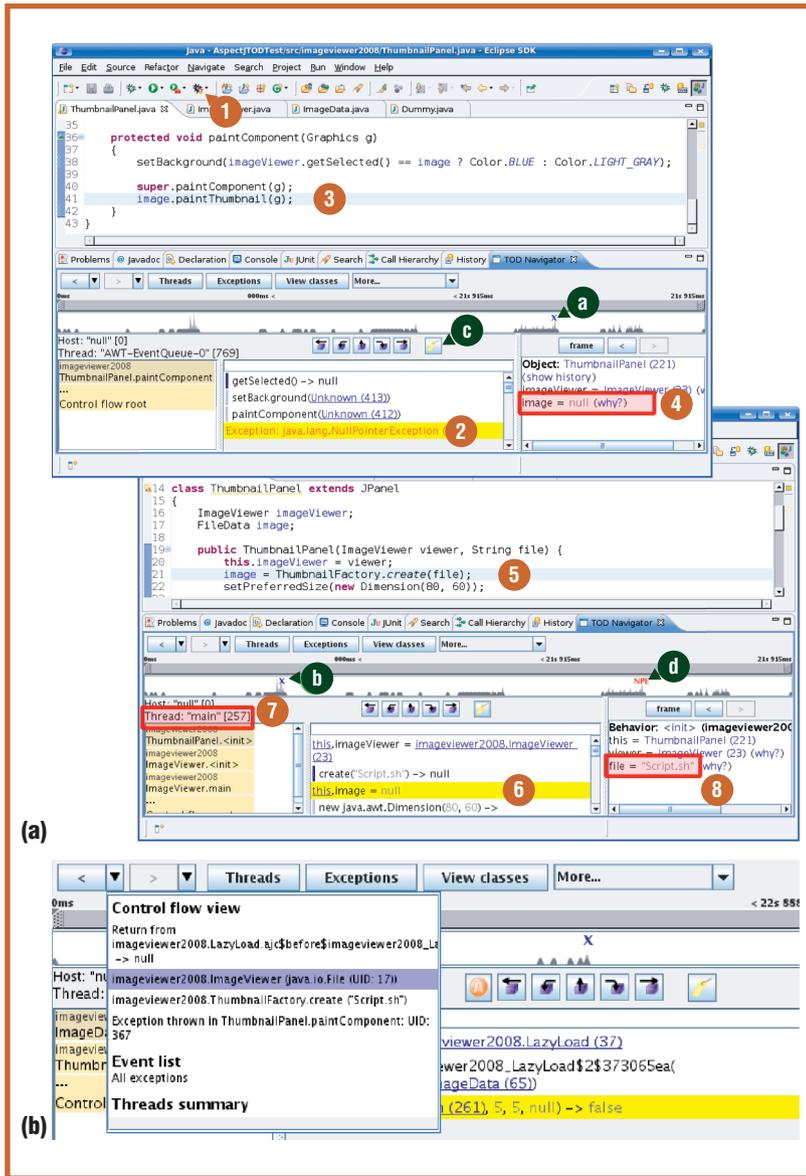


Figure 3. TOD (trace-oriented debugger) user interface. (a) Hunting for the root cause of a NullPointerException using the “why?” link: going from the symptom to the cause in eight simple steps. (b) Navigation history. Web browser-like back and forward buttons permit users to easily navigate between visited events and views.

Debugging with TOD

TOD supports temporal navigation via stepping both forward and backward in time. In addition, it supports fast causal navigation via a link displayed next to the value of inspected variables, which lets the user directly jump to the event that assigned the variable its current value. We now

describe a bug-hunting session that uses this feature, as illustrated in Figure 3.

After launching the buggy program with the TOD launch button (1), we can easily locate the exception event in the execution trace. Once the exception event is selected in the main control flow view (2), the corresponding source code line is automatically highlighted (3). Here we notice that the thumbnail field of the current `ThumbnailPanel` object is indeed null (4), which is why the exception was thrown. Clicking the “why?” link (4) immediately brings us to not only the source code line where the value of the field is set (5) but also to the precise event that caused this particular assignment (6). Note that the assignment occurred in a different thread than the one that threw the exception (7). Inspecting the program state at the newly selected event shows that we tried to create a thumbnail of a `.sh` file (8), which failed.

In this simple example, TOD let us jump in just a few steps directly from the bug’s symptom (the exception) to its cause (the mishandling of nonimage files). The same bug hunting with a breakpoint-based debugger would have been tedious because the program potentially has many places in which the thumbnail field is set apart from the constructor and many correct instantiations of `ThumbnailPanel` to step through.

Such a toy example doesn’t show TOD’s full potential, however. Although it would be too lengthy to relate here, we’ve used TOD to quickly solve difficult problems, such as bugs in the TOD database itself, and to understand issues that arose in our use of highly complex software such as the AspectBench Compiler for AspectJ (<http://abc.comlab.ox.ac.uk>).

Not Getting Lost: Bookmarking to the Rescue

Given the huge amount of events that TOD can record, it’s crucial to help users avoid getting lost while navigating an execution trace. To this end, TOD lets users bookmark events and objects, and lets them quickly access previously visited locations.

A timeline above the main TOD view displays bookmarked events. Additionally, the event that’s currently selected in the main view also appears in this timeline, so users can immediately find their way around in the execution trace relative to known landmarks. This is particularly useful when using the “why?” link, which can lead to events that occurred far in the past, in completely different contexts.

The green balloons in Figure 3a illustrate the

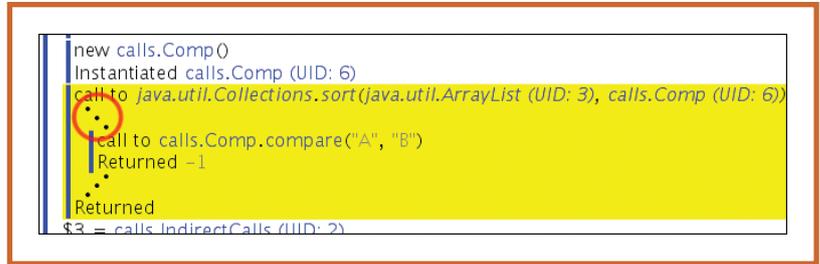


Figure 4. TOD (trace-oriented debugger) displays missing information while reconstructing control flow. Making the missing information explicit allows the user to reason soundly about the information that’s available.

process of event bookmarking. The position of the current event is always marked in the timeline (a, b). When users feel they reached an important landmark (or a starting point for exploring several program paths), they press the bookmark button (c), which also lets them choose a name and color for the event (d).

Users can also bookmark and assign colors and names to individual objects. It’s therefore possible to mark an object involved in a failure so that previous usage of that object is easy to spot during navigation.

In addition to bookmarking, the user interface provides back and forward buttons similar to a Web browser’s that permit access to the entire navigation history (see Figure 3b).

Support for Partial Traces

Although we designed TOD to support huge execution traces, it isn’t always practical to record every single event: the runtime overhead caused by event generation is considerable, as are the storage requirements. Instead, because only certain parts of a program execution are of interest, users can capture *partial traces*.⁵

Developers obtain partial traces in TOD by using both *static* and *dynamic scoping*: static scoping consists of selecting which classes should or shouldn’t generate events. Dynamic scoping consists of enabling or disabling trace capture at runtime, either by using a simple API directly in the debugged program or by using a switch button in the debugger front-end. Dynamic scoping is particularly useful when a bug occurs after a long running time or under specific dynamic conditions. For instance, in a Web application, it might be interesting to limit trace capture to the processing of a particular HTTP request’s control flow.

The downside of partial traces is that they are, indeed, partial. Consequently, developers can’t reconstitute every part of the debugged program’s history. To allow them to soundly reason about available information, TOD systematically makes missing information explicit. For instance, in Figure 4, the small dots indicate that control flow information is missing: between the execution of `sort`

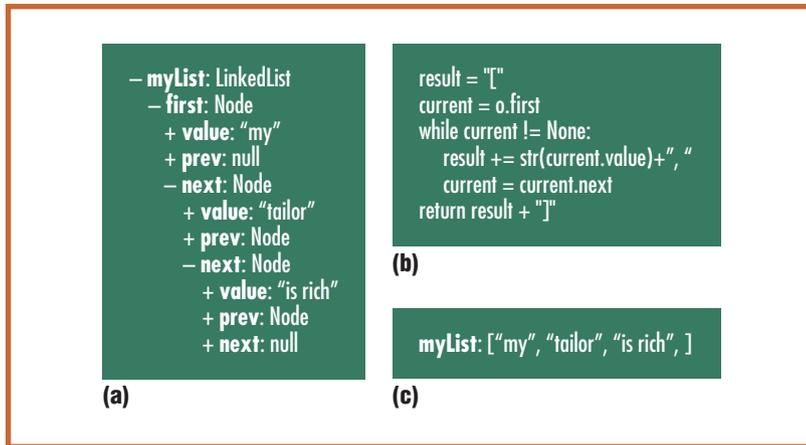


Figure 5. Custom formatter for a linked list. (a) The default formatting shows the list’s internal structure. (b) The custom formatter receives the object to process in the *o* variable. It iterates through the linked list using its internal structure (the list header and each node’s next pointer) to (c) construct a string representation.

and compare, some unrecorded computation took place because the standard `Collections.sort` method wasn’t included in the trace.

In practice, the benefits of partial traces far outweigh the drawbacks. Combining static and dynamic scoping has proven invaluable for debugging long-running, CPU-intensive programs such as the TOD database itself.

Custom Formatters

Similar to most breakpoint-based debuggers, TOD displays reconstituted objects by default as a list of *field = value* pairs and lets the user explore the object graph by “opening” objects of interest. Although this is sufficient in some cases, a higher-level representation is more useful in others. Consider, for instance, a linked list (see Figure 5): the user is usually more interested in the list’s sequence of elements than in details of each node’s next and previous pointers.

Like modern IDEs such as Eclipse, TOD lets the user define custom formatters for classes of interest. In TOD, these formatters are small scripts that can easily access the fields of reconstituted objects to produce rich (HTML) textual representations. Figure 5 shows an example of such a formatter.

Current State of the Practice

Most modern IDEs only provide a breakpoint-based debugger out of the box, and they all have roughly the same set of features: setting regular or conditional breakpoints, watchpoints, forward stepping, and the ability to inspect the current stack frame and the objects reachable from it.

However, some omniscient debuggers are available today.

Omniscient Debuggers for Java

ODB is one of the first omniscient debuggers for Java.³ Like TOD, it obtains execution traces by instrumenting classes as the JVM loads them; however, ODB stores the captured trace data inside the target JVM. This raises some issues: the available heap space limits the amount of trace data, and references to objects that are no longer in use are kept, preventing proper garbage collection. A unique feature of ODB is the ability to resume execution from any point in time with a modified state.

The Whyline for Java lets users select questions about why some behavior did or didn’t occur.⁶ The Whyline automatically generates these questions based on a combination of static and dynamic analysis. It can deal not only with the program’s internal state—for example, “why does variable *x* have value *y*?”—but also with its textual and graphical output, all the way down to individual pixels. Although the Whyline can analyze relatively large execution traces (for example, 35 million events), its scalability is limited because it performs the analysis in memory.

JIVE is an interactive visualization environment for Java programs⁷ that provides UML-like sequence diagrams as well as object diagrams extended with information about the current method call. Users can reduce the diagrams’ level of detail to fit more information on the screen, but it isn’t clear that this mechanism scales past a few hundred elements. JIVE supports forward and backward stepping but not quick causal navigation. It captures execution trace using JPDA, JVM’s debugging interface, and processes it in memory, thus limiting the system’s efficiency and scalability.

Other Platforms

Omniscient debuggers have also been proposed for platforms other than Java.

Lisp. In 1984, ZStep provided a reversible stepper for Lisp that allowed developers to step forward and backward and see the result of evaluated expressions in parallel to the corresponding source code.⁸ Its sequel, ZStep95, added the ability to relate graphical output to the event that caused it, as well as tape-recorder-like controls for easier navigation.⁹ However these systems didn’t handle side effects, causal links (except for graphical output), or scalability issues.

	Platform	Mechanism	Storage media	History size	Runtime overhead	Partial traces	Casual nav.	High-level overviews	IDE integration
TimeMachine	Embedded	?	RAM/probe	1e9	Soft.: ? Hard.: none	?	?	✓	Part of Green Hill's MULTI IDE
UndoDB	Linux	Checkpoint replay	RAM	n/a	7x	Not applicable	✗	✗	Wrapper for gdb
Chronicle	Linux	Event log	Disk	1e9	300x	?	✓	✗	Plugin for Eclipse CDT
[Lienhard]	Squeak	Event log	RAM	1e5	6x	Events on unreachable objects are discarded	✗	✗	Integrates into
Unstuck	Squeak	Event log	RAM	1e5	250x	Lexical scoping	✗	✗	Integrates into the platform
Whyline	Java	Event log	Disk	1e7	252x/20x	Lexical scoping	✓	✗	No
JIVE	Java	Event log	RAM	?	?	Lexical scoping	✗	✓	Plugin for Eclipse JDT
ODB	Java	Event log	RAM	1e6	95x/37x	Lexical scoping	✓	✗	Limited Eclipse integration
TOD	Java	Event log	Disk	1e9	83x/28x	Lexical & dynamic scoping Missing info explicit in GUI	✓	✓	Plugin for Eclipse JDT/AJDT

History size gives the order of magnitude of the number of events that can reasonably be collected and processed by the system.
 Runtime overhead gives an idea of the slowdown caused by the debugger.
 In the Partial traces column, lexical scoping means that it's possible to select the classes or packages to instrument, and dynamic scoping means that it is possible to activate or deactivate trace capture at runtime.
 Causal navigation indicates if the debugger permits to directly jump to the past event that set a variable to its current value.
 High-level overviews indicate if the system can provide summary views of the debugged program.

Figure 6. Comparison of available omniscient debuggers. For systems on which we performed our own benchmarks, we provide two figures: the first is the runtime overhead in the worst case (that is, for a fully instrumented, CPU-intensive program), and the second corresponds to a more typical situation (a run of the jTidy HTML beautifier program). For the other systems, the unique figure is the one provided by the system's authors.

Native. TimeMachine by Green Hills Software (www.ghs.com/products/timemachine.html) is an omniscient debugger for embedded systems (PowerPC, ARM and similar architectures). On some platforms, a specialized hardware probe lets developers capture trace data without incurring any runtime overhead; otherwise, it uses traditional software instrumentation. In addition to the usual features of omniscient debuggers, developers can also use TimeMachine as a profiling tool.

UndoDB is an omniscient debugger for native x86 Linux programs by Undo (www.undo-software.com). As opposed to most of the other tools presented here, UndoDB is based on a checkpoint/replay mechanism: it periodically obtains a *checkpoint*, or *snapshot*, of the process memory, and uses a replay technique to reconstruct the program's state between checkpoints. This mechanism yields a relatively low runtime overhead, but it doesn't allow causal navigation.

Chronicle (<http://code.google.com/p/chronicle-recorder>) is an open source omniscient debugger for native x86 Linux programs with an architecture similar to that of TOD: binaries are instrumented so that they send trace data to an out-of-process, disk-based database. A key characteristic of Chronicle is the aggressive compression and indexing of events that lets developers efficiently record very large traces and process queries.

Smalltalk. Unstuck is an omniscient debugger for Smalltalk that's similar in architecture and operation to ODB.¹⁰ Adrian Lienhard and his colleagues¹¹ proposed another back-in-time debugger for Smalltalk that handles the scalability issue by using partial traces but in a very different way from TOD. They postulate that information about objects that aren't reachable at a certain point in time (that is, objects eligible for garbage collection) can be discarded. Although discarding this information boosts efficiency, we think that a bug's root cause can have occurred in the context of objects that have been discarded long before the bug's symptoms manifest themselves, thus rendering this approach ineffective in some cases.

Comparison

Figure 6 summarizes the characteristics of these tools; TOD has several characteristics that make it a competitive alternative:

- TOD's scalable database engine enables fast storing and querying of events. Moreover, it can be distributed over a cluster of machines to further improve its scalability.
- The support for partial traces dramatically enhances TOD's applicability by offering expressive means to specify selective trace generation and adequately report incomplete information.

About the Authors



Guillaume Pothier is a doctoral candidate in the Computer Science Department of the University of Chile. His main research interests are programming languages and tools. His current research is dedicated to finding ways and means to make omniscient debugging practical. He obtained a MEng in computer science from the École des Mines de Nantes, France. He is a member of the ACM. Contact him gpothier@dcc.uchile.cl.

Éric Tanter is an assistant professor in the Computer Science Department of the University of Chile, where he leads the PLEIAD (Programming Languages and Environments for Intelligent, Adaptable, and Distributed Systems) laboratory. His research interests include programming languages and tool support for modular and adaptable software. Tanter received his PhD in computer science from both the University of Nantes and the University of Chile. He is a member of the ACM and the IEEE. Contact him at etanter@dcc.uchile.cl.



- The GUI's responsiveness, achieved through efficient query processing, lets users interactively navigate huge execution traces.
- TOD's specialized GUI metaphors, such as the “why?” link, bookmarks, and timelines, allow for effective navigation and program understanding.
- The tight Eclipse integration lets users smoothly integrate TOD in the development process (integration with IntelliJ and NetBeans is currently ongoing).

On the other hand, some of the features provided by other systems are missing from TOD:

- Whyline lets users ask negative questions such as, “Why did method *x* not execute?” These questions frequently occur during the debugging process.
- Whyline lets users relate the program's textual and graphical output to the event that caused them. Support for textual output in TOD is underway, but support for graphical output would require considerable work.
- TOD has a runtime overhead similar to that of ODB and Whyline, yet it provides a much greater scalability. However, systems such as UndoDB and Lienhard's have a much lower overhead, and TimeMachine has no overhead at all because it uses a hardware probe. We strive to reduce TOD's runtime overhead by using static analyses to limit the amount of redundant information it captures.
- JIVE provides graphical visualizations of the object graph, which can be very useful for program understanding. Although TOD supports custom formatters, they're only textual.

Although TOD is a prototype and still contains many rough edges, we found it invaluable in our day-to-day development experience, both in academia and in industry. If you want to give it a try, it's free and open source (<http://pleiad.cl/tod>). Don't hesitate to subscribe to the mailing list or to contact us for more information.

Although omniscient debugging seems to be slowly attracting more attention in industrial settings, new programming languages and paradigms present new challenges. We can already identify three major areas in which the development of practical omniscient debugging is crucial:

- Dynamic languages (for example, Python, Ruby, and so on) are becoming increasingly popular. Improving the debugging support could help alleviate, at least to some degree, the lack of static type checking.
- Developing concurrent and distributed systems is notoriously difficult, in particular because failures can be hard to reproduce. Being able to automatically record and later navigate through the execution history of such programs is thus of primary importance.
- Because it adds more possible loci for late binding, aspect-oriented programming (AOP) makes it more difficult for programmers to mentally reconstruct a program's execution flow.¹² Appropriate development tools—in particular debuggers—are required to support AOP.

We're already exploring how omniscient debugging can provide adequate support for AOP.¹³ We're also developing a version of TOD for Python. Concurrent and distributed programming is on our research agenda.

Because omniscient debugging is such an effective tool for program understanding, it greatly enhances the software development process. It's therefore crucial to devote efforts to make it practical and applicable in as many situations as possible and address the different challenges to its adoption. ☺

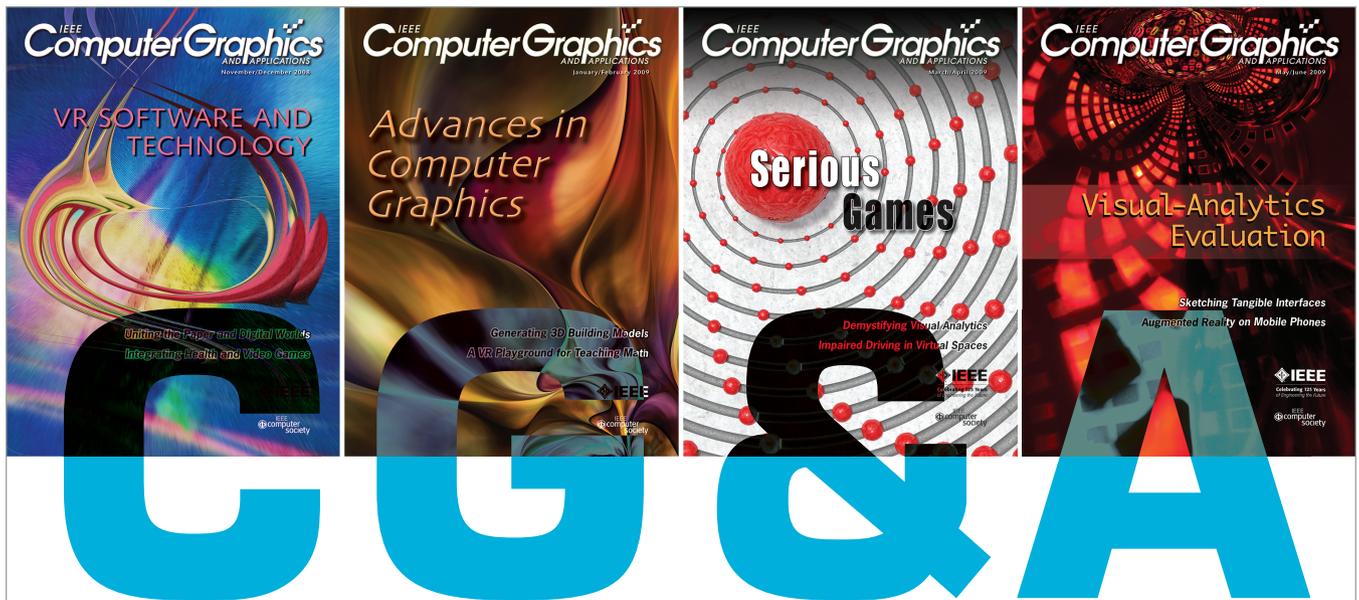
Acknowledgments

We thank Greg Law of Undo for providing detailed technical information on UndoDB, as well as Alexandre Bergel, Johan Fabry, Adrian Lienhard, Olivier Motelet, and the anonymous reviewers for their valuable comments.

References

1. Nat'l Inst Standards and Technologies, "Software Errors Cost U.S. Economy \$59.5 Billion Annually," June 2002; www.nist.gov/public_affairs/releases/n02-10.htm.
2. M. Eisenstadt, "My Hairiest Bug War Stories," *Comm. ACM*, vol. 40, no. 4, 1997, pp. 30–37.
3. B. Lewis, "Debugging Backwards in Time," *Proc. 5th Int'l Workshop Automated Debugging (AADEBUG 03)*, M. Ronsse and K.D. Bosschere, eds., Computer Science Repository, 2003.
4. R.M. Balzer, "EXDAMS—Extendable Debugging and Monitoring," *Proc. Am. Federation of Information Processing Societies Spring Joint Computer Conf.*, ACM Press, 1969, pp. 567–580.
5. G. Pothier, É. Tanter, and J. Piquet, "Scalable Omniscient Debugging," *Proc. 22nd ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages and Applications (OOPSLA 07)*, ACM Press, 2007, pp. 535–552.
6. A.J. Ko and B.A. Myers, "Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior," *Proc. 30th Int'l Conf. Software Eng. (ICSE 08)*, ACM Press, 2008, pp. 301–310.
7. P. Gestwicki and B. Jayaraman, "Methodology and Architecture of JIVE," *Proc. 2005 ACM Symp. Software Visualization (SoftVis 05)*, ACM Press, 2005, pp. 95–104.
8. H. Lieberman, "Steps toward Better Debugging Tools for Lisp," *Proc. 1984 ACM Symp. LISP and Functional Programming (LFP 84)*, ACM Press, 1984, pp. 247–255.
9. H. Lieberman and C. Fry, "ZStep 95: A Reversible, Animated Source Code Stepper," *Software Visualization—Programming as a Multimedia Experience*, J. Stasko et al., eds., MIT Press, 1998, pp. 277–292.
10. C. Hofer, M. Denker, and S. Ducasse, "Implementing a Backward-in-Time Debugger," *Proc. Net.Object Days (NODE 06)*, Lecture Notes in Informatics, vol. P-88, Gesellschaft für Informatik, 2006, pp. 17–32.
11. A. Lienhard, T. Gírba, and O. Nierstrasz, "Practical Object-Oriented Back-in-Time Debugging," *Proc. European Conf. Object-Oriented Programming (ECOOP 08)*, Springer, 2008, pp. 592–615.
12. T. Elrad, R.E. Filman, and A. Bader, "Aspect-Oriented Programming," *Comm. ACM*, vol. 44, no. 10, 2001, pp. 29–32.
13. G. Pothier and É. Tanter, "Extending Omniscient Debugging to Support Aspect-Oriented Programming," *Proc. 23rd ACM Symp. Applied Computing (SAC 08)*, vol. 1, ACM Press, 2008, pp. 266–270.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.



IEEE Computer Graphics and Applications bridges the theory and practice of computer graphics. From specific algorithms to full system implementations, CG&A offers a unique combination of peer-reviewed feature articles and informal departments. CG&A is indispensable reading for people working at the leading edge of computer graphics technology and its applications in everything from business to the arts.

Visit us at www.computer.org/cga