ISSN 1346-5597

National Institute of Informatics

NII Technical Report

Visualization of Concurrent Program Executions

Cyrille Artho, Klaus Havelund, and Shinichi Honiden

NII-2007-006E
May 2007

Visualization of Concurrent Program Executions

Cyrille Artho
Research Center for Information Security (RCIS),
National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan

Klaus Havelund
NASA Jet Propulsion Laboratory/Columbus Technologies, Pasadena, USA

Shinichi Honiden
National Institute of Informatics, Honiden Laboratory, Tokyo, Japan

Abstract

Various program analysis techniques are very efficient at discovering failures and properties. However, it is often difficult to evaluate results, such as program traces. This calls for abstraction and visualization tools. We propose an approach based on UML sequence diagrams, addressing shortcomings of such diagrams for concurrency. The result is a more expressive visualization that can provide all the necessary information at a glance.

1. Introduction

Certain program analysis techniques work directly on the executable program. For instance, *run-time verification* monitors executions of (possibly concurrent) programs [2, 8, 10, 23]. *Software model checking* also analyzes executions of concurrent systems, producing an error trace when a failure is found [3, 5, 12, 27]. Tool capabilities have advanced, but their outputs still consist of overly concise reports, or very long program traces. Hence, understanding the nature of failures and properties remains difficult. Program traces are a widely used way to show how a program behaves up to a given point, but may grow very large. *Abstractions* can simplify program traces; indeed, a typical trace shown to the end user contains mostly method calls and thus constitutes a useful abstraction. For sequential programs, a program trace or even a stack trace (a subset of the entire program trace) contains enough information for a concise and useful summary.

However, large or concurrent traces are hard to read. In a concurrent program, context switches interrupt threads. By showing only a thread ID prior to each step, a program trace has no clear visual indication of such a context switch. Furthermore, it is not clear whether a context switch is necessary to reproduce a failure, or whether it just happened to be part of the schedule executed that lead to a failure. In order words, the happens-before relation between events [17] is not visible in a program trace, even if it may be available from data gathered at run-time [8].

Program trace visualization addresses the problem of understanding dynamic program behavior. Two approaches exist: *still visualization,* where *all* events are visualized in one view, and *animations.* Still visualization includes UML sequence diagrams [22] and plots of event sequences, such as in [21] or a large number of similar tools. Common notations such as UML are not capable of capturing asynchronous events [15]. Animations use either a two-dimensional view of each state [5, 6], or a three-dimensional animation [20]. In animations, the order in which events occur is intuitively visible; however, an animation also imposes a total order on concurrent events where only a partial order may exist.

There seems to be a relationship between still visualization and automated gathering of requirements [7, 9, 28], where a requirements specification of a program is extracted from one or more program runs. As an example, a state machine extracted from several runs can be regarded as a still visualization of the program's behavior as well as a specification of its behavior during those runs. Extraction of such specifications from runs can serve as oracles for later runs, for example for use in regression testing, or simply as a means of program understanding. Other forms of less visual specifications can be extracted, such as for example temporal logic specifications [28]. Such specifications also have natural visualizations, for example as time lines [24].

This paper is organized as follows: Section 2 describes our visualization approach. Reasons for design choices in our visualization are given in Section 3. Section 4 shows a more complex example, where our visualization approach was used to elucidate a complex error trace. Section 5 discusses implementation issues. Section 6 concludes and outlines challenges ahead.

2. Our visualization approach

In still visualization, even complex event chains can be visualized "at a glance". We chose an approach based on UML sequence diagrams [22] because UML diagrams are fairly widely accepted in industry and supported by tools. UML sequence diagrams capture sequences of method calls, but cannot deal with concurrency. We have therefore extended UML sequence diagrams in several ways to include the missing features required to visualize concurrent events.

2.1. Limitations of UML sequence diagrams

Sequence diagrams are designed to show sequences of method calls. This task is closely related to displaying a program trace. UML sequence diagrams have been studied extensively and defined precisely [15, 18]. Our work expands on existing sequence diagrams and gives them a meaning in concurrent scenarios.

Our initial approach is based on previous extensions of UML sequence diagrams for clarifying the current execution context [18]. Previous work [18] has not addressed concurrency. In particular, UML sequence diagrams cannot illustrate the following:

- A Thread as data structure *and* executable task.
- "Invisible" task switches induced by the thread scheduler.
- Activations and suspensions of threads. In most modern programming languages that follow a POSIX thread model [13, 19, 25], a thread is inactive when created. Once a special method (such as `start`) is called, it becomes active, but can be suspended, through actions that wait on events (such as termination of another thread, or notification of a change of a shared conditional).
- Time-based suspension. A thread can suspend itself ("sleep") for a certain time, allowing other threads to run. The same effect can be induced by the thread scheduler through a context switch. Its occurrence is therefore somewhat arbitrary, and cannot be used for reliable synchronization of events. We have therefore chosen not to visualize this artifact.
- The *happens-before relation* [17]. This relation indicates that certain events must happen strictly before another event occurs. For instance, any events leading to the creation and activation of another thread must happen before actions of the child thread take place. This is obvious as the child thread did not exist during such previous actions. However, when a large number of such events occurs, understanding of the happens-before relation is often non-trivial, and should therefore be included in a visualization.[1]
- Locking. Many programming languages use locks for mutual exclusion [13, 19, 25]. The presence of locking actions may delay a thread until a certain lock is available. This is partially reflected by the happens-before relation. For conciseness, we have not added another mechanism to visualize locking and lock sets.

The happens-before relation states, informally, that based on observed events, certain reorderings of events are possible. Given events would still occur with an equivalent global program state after each event, and the overall outcome of the program would not be changed. More formally, if events are reordered within the happens-before relation, an observer that evaluates global program states always sees the same sequence of global program states, even though invisible internal actions can be ordered in different ways [17]. This resulting property is called *sequential consistency.*

2.2. Our UML extensions

Our visualization addresses the concerns described above. It is based on the Java programming language, but readily applicable to other programming languages

[1]Until recently, with common run-time verification algorithms, the knowledge of this relation was often incomplete. A recent algorithm computes this relation precisely without much overhead [8].

using the same thread model [13, 19, 25]. Our visualization distinguishes between the two roles of a Java thread as an executable task and a data structure [13]. The thread data structure holds information such as thread name and ID, and can be extended with other data. A thread as a *task* constitutes a light-weight process that shares the global heap with other threads. This article refers the following methods of the Java API to denote crucial operations on threads and locks:

- method `start` causes a thread to begin execution;
- `join` suspends the current thread until the target thread has terminated;
- `wait` suspends the current thread until another thread issues `notify` or `notifyAll`.[2]

Threads as a data structure are visualized like other object instances in UML sequence diagrams. Our first extension is the visualization of role of a thread as an executable task by a hexagon. A dashed arrow pointing to the left symbolizes the thread scheduler running a thread (task). As in UML sequence diagrams, solid arrows depict a method call or return, and solid squares show a method being executed.

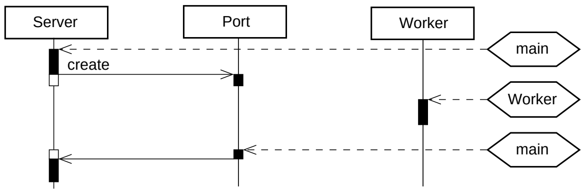

Figure 1. A thread switch from main to Worker, and back.

Figure 1 includes these basic elements. It shows the illustration of *context switches* between threads. At the beginning of the scenario, the *main* thread is scheduled. This thread creates a new instance of *Port.* During the call to the constructor, the scheduler switches to another thread, *Worker*. The interruption of the *main* thread is shown by a gap in the time line of the call from *Server* to *Port.* Thread *Worker* executes for a certain amount of time without making any method call, after which the *main* thread is scheduled again, and the method call to *Port* completes.

[2]This simplified definition holds if one thread is waiting on a shared lock. For the complete definition that covers multiple waiting threads, refer to the language specification [13].

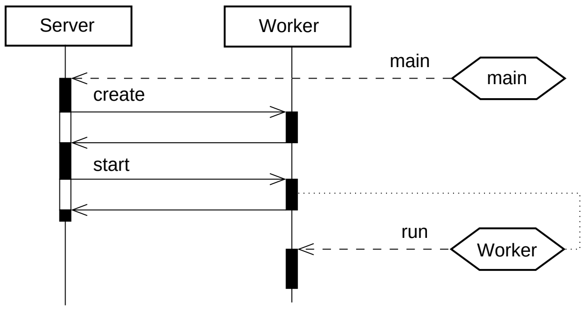

Figure 2. Thread creation and start.

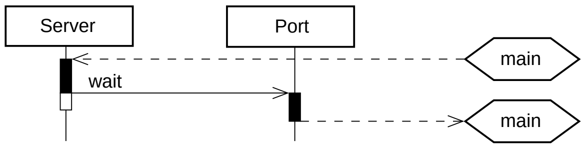

Figure 3. Thread suspension using wait.

Dotted lines show event dependencies according to the happens-before relation [17]. If there is a dotted line from a point *p* to a hexagon *t*, then any events following an activation of thread *t* could have started right *after p*. Figure 2 shows the happens-before relation based on a slightly more complex example, where a worker thread is *started* by the *main* thread. At the beginning of the program, the *main* thread is scheduled, as depicted by a hexagon. A dashed arrow points to the beginning of the sequence of actions of that thread, symbolizing scheduling of actions of this thread. Creation of thread *Worker* involves initialization of the data structure and is no different from initializing a normal object. The thread is started by a library call, which interfaces with the operating system. Any actions of thread *Worker* can occur at any time after this point, symbolized by the dotted line. In other words, actions of thread *Worker* could be moved up to the top of the horseshoe-shaped dotted line.

The start of a thread is shown by a corresponding action in the thread scheduler, using an dashed arrow pointing from a hexagon to the left. Likewise, thread *suspension* is depicted by such a dashed arrow pointing to the right, from the lower part of the black box denoting a method call, to the thread being suspended. In Figure 3, the *main* thread runs and calls `wait` on lock *Port*. The arrow originates from the end of the method call rather than its middle because the current thread still executes instructions up to its suspension.

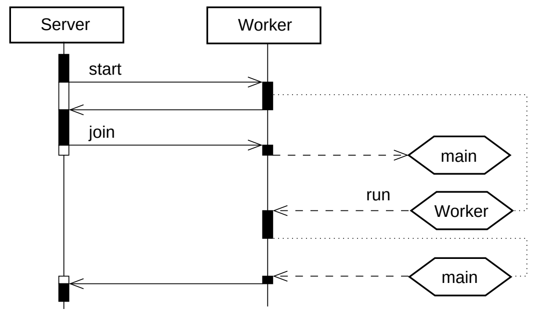

Figure 4. Thread suspension using join.

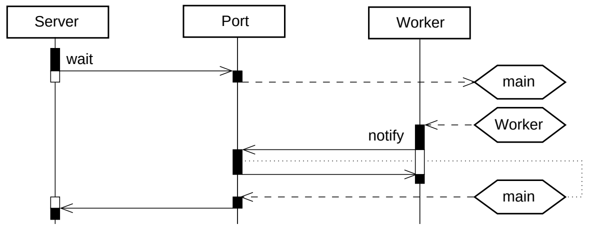

Figure 5. Thread notification.

Unlike thread suspension, thread *termination* is not shown as a scheduler event. Because no further actions of that thread exist, there is no compelling need to decorate thread termination. On the other hand, thread termination may influence the behavior of other threads waiting on that event, and thus contribute to the happens-before relation. Figure 4 shows an example involving `Thread.join`. As in subsequent figures, some initial thread activations have been omitted for brevity. Thread *main* starts a worker thread and waits upon its termination using `join`. This suspends *main* until *Worker* terminates. Any events in the *main* thread following that `join` call can only happen after Thread *Worker* has terminated, as illustrated by the dotted line.

Thread *notification* is similar to re-activation of a thread after suspension. In the previous example involving `join` and thread termination, one event leads to thread suspension (`join`), while another event (thread termination) allows the suspended thread to continue. The same pattern exists for `wait`/`notify`, the key difference being that continuation of the suspended thread is achieved by a special call (`notify`) rather than termination of another thread.

Figure 5 shows an example for `wait`/`notify`. As in Figure 4, suspension of the waiting thread is shown by a dashed arrow pointing to the right. Here thread *main* waits on *Port*, which is used as a lock and semaphore according to standard Java semantics [13]. After suspension, thread *Worker* is scheduled, which notifies all threads waiting on *Port*. Notification leads to activation of one of the suspended threads (*main* in the example). Once notified, a thread is again ready to run, as shown by the happens-before relation. Activation is takes place inside native method `notify`.

Notification can target a single thread, or *all* threads waiting on a lock, using `notifyAll` in Java. Whenever several threads wait for the same lock, notification will enable all of them to run. In this case, the happens-before relation concerns multiple threads. Furthermore, it is often the case that only a single thread will continue to execute, while all the other threads re-check a shared condition and then go back to being suspended by calling `wait` again.

Figure 6 depicts such a scenario. At the beginning of the situation shown, threads *Worker 1* and *Worker 2* are waiting on lock *Port*. Thread *main* calls `notifyAll` on that lock, whereupon *Worker 1* is scheduled first. That thread can complete an action on global data (e. g., consuming a shared resource, such as a connection from a client). After that, the scheduler runs *Worker 2*. In the example, the shared resource has been consumed by *Worker 1*, so *Worker 2* has to wait again until another thread makes the resource in question available again. Therefore, *Worker 2* subsequently waits again after re-checking its condition. This allows the scheduler to execute *Worker 1* again.

3. Design decisions

Our extension of UML sequence diagrams maintains a close and concise mapping [14]. We address all commonly available concurrency artifacts [13, 19, 25], using four new symbols. First, we distinctly express the role of a thread as a task. Second, we make task activations and context switches visible. The hexagon as a task symbol is visually clear. Furthermore, it allows attachment of arrows denoting thread context switches, and lines representing the happens-before relation. Locks are not directly visualized, but can be shown by secondary notations, such as annotations.

Third, thread suspension is different from a normal context switch (where a thread can continue to run again later). We chose to represent this with a symbol that is the reverse of thread activation by a context switch. We believe that this is consistent.

Finally, the happens-before relation [17] explains possible event orderings. It is visualized by dotted

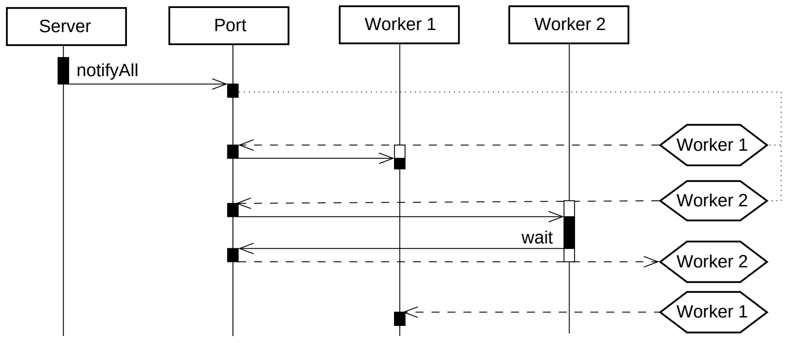

Figure 6. Thread notification: Main thread notifies both worker threads.

lines. Events are not totally ordered [17]. Thus, more constraining visualizations, such as shaded regions, fail for more complex scenarios.

We chose to illustrate calls to `wait` and `notify` like any other method calls, by a solid black box. This does not only provide consistency, but also allows for a better illustration of the side effects of these methods.

The precise timing of thread activations cannot be determined, as it occurs inside special method calls, such as `start` and `notify`. Hence, the line visualizing the happens-before relation is placed in the middle of such method calls. Thread suspension via `join` is different, as its argument (the thread in question) actually has to terminate before said call returns. Therefore, the line of the happens-before relation must be attached to the bottom of the box, representing completed method execution, which implies thread termination.

Method calls to `wait` do not affect the happens-before relation. This is because `wait` has no direct effect on other threads, so any events of other threads are not correlated to when the current thread is suspended.

We chose not to visualize locking and lock sets directly. Inclusion of lock sets may be done by annotations, but will decrease conciseness of the graph. Likewise, atomicity of actions, which depends on locking, is not shown. While correct lock usage corresponds to a "hard mental operation" [14], our visualization captures the key problems in concurrency on a slightly higher level of abstraction, improving scalability.

4. Trace visualization

The example application was subject to previous research [1, 4]. It implements a chat server, where multiple clients can connect and interact with each other. The architecture of the chat server is fairly complex, involving a main server thread to accept connections and one worker thread per connection. Worker threads use shared data structures to send a message to all other clients (see Figure 7). This architecture is comparable to modern web servers [11].

In the chat server, the main server thread listens to incoming connections and creates a worker thread for each connection that is handled. These worker threads use shared data structures to send a message to all other clients. The main thread inside each client process is the only thread and therefore not shown separately. Because the server contains several threads, the main thread is listed as such. Each message from a client is handled by the corresponding worker thread on the server side. This worker thread then sends that message to each other client by accessing the shared array that contains references to all the other workers. Through this array, the sockets connecting the chat server to each client can be retrieved. Therefore any interaction between clients always occurs via the server application, where it is handled by a particular worker thread.

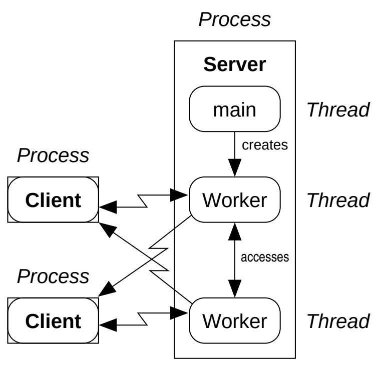

Figure 7. Chat server architecture.

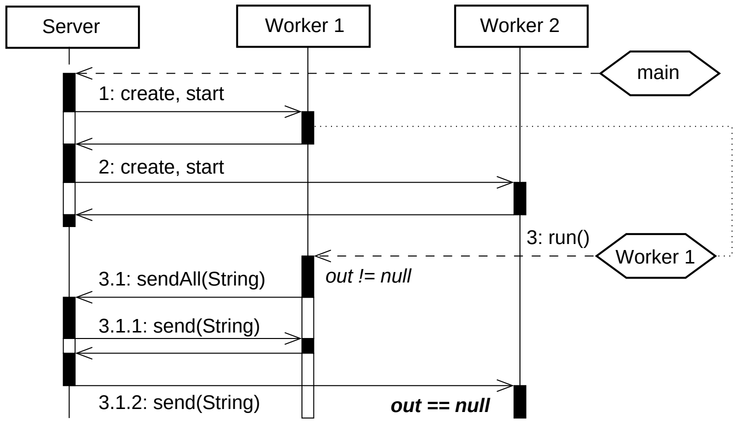

Figure 8. Complex interaction illustrated by our extended sequence diagram.

The difficulty of understanding program semantics became obvious when analyzing a complex error trace containing six threads and hundreds of transitions between threads [1]. We subsequently sought visualization techniques. Figure 8 shows the essence of such an error trace. Trace abstraction was performed manually [26], but we believe that this can be automated in a scalable way. Crucial events include thread creations and changes in the predicate that makes the program fail (field *out* being `null` when accessed). The program trace consists of the three classes (Server, Worker 1, Worker 2). The Server class first initializes the two worker threads (using them as data structures), and enables them to run (as threads). In the example, the second client then proceeds to send a message. The second worker thread is subsequently scheduled to handle this event. This entails sending a message to all clients. To achieve this, the worker thread accesses data structures of all worker threads, including itself. When accessing field *out* of the other Worker object, it accesses an uninitialized field, producing a `NullPointerException`.

The failure occurs because the constructor of the worker threads initializes only some fields. Initialization of reference *out* is performed in the `run` method of a worker thread, rather than in its constructor. Therefore, a particular sequence of initialization and execution leads to a null pointer dereference in one of the two threads involved. Annotations about *out* (in italics) show changes and dereferences inside a method call. The happens-before relation links initialization of thread Worker 1 to its execution, showing that this thread could have been scheduled after the other worker thread was initialized. Such a schedule would have avoided a failure. The actual schedule shown in this trace is different, leading to a failure. The error trace in its entirety is rather long and difficult to read. For readers with some experience in the Java socket API, Appendix A includes the full error trace, and an explanation of each step.

5. Visualizer architecture

As described earlier, events can be contained in an error trace of a model checker, or be generated at runtime. Figure 9 shows how events are extracted in both cases. In model checking (MC), the resulting error trace is visualized. In run-time verification (RV), event generation has to be embedded into the program being analyzed. This can be done with automated code instrumentation, for example, using aspect-oriented programming [16].

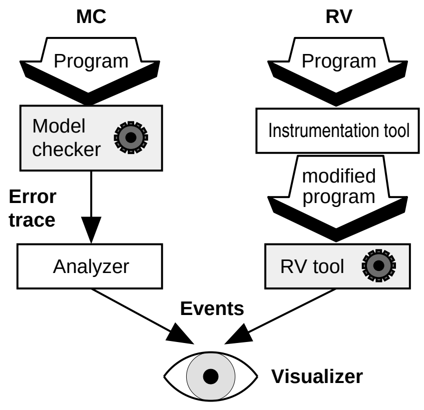

Figure 9. Event extraction / visualization.

The modified program will, in addition to its normal functionality, emit events to our visualizer. In RV, the visualizer can operate on-line, using live events, or off-line, after termination of the program. Error traces from model checkers are only examined off-line, after termination of the model checker.

The visualization module can be built as a library. Calls can be made at points defined by a programmer, or at automatically instrumented places. Alternatively, a parser can be built for a particular input format, either reading error traces from a model checker, or reading logged execution traces. The result of the parse can then be visualized with the same package, independently of the application domain.

6. Conclusions and future work

Understanding a concurrent program trace is difficult. Still visualization builds on trace abstraction and shows the essence of a trace. Our approach builds on UML sequence diagrams and can illustrate a failure on a complex error trace clearly, and can also be used as a tool to reverse engineer program behavior. Error traces may originate from a model checker or a run-time verification tool and can be visualized in the same way.

Future challenges include automated tool support, so our visualization can be applied to larger examples. This will also allow us to explore the scalability of our visualization when used with different abstraction or exploration techniques. We will also consider visualization of timeouts and locks through means other than annotations.

A. Full error trace

Figures 10, 11, and 12 contain the full error trace of the example application. From the raw output of Java PathFinder 3.0a (JPF) [27], the following modifications have been performed:

- Steps showing execution of code inside Java library classes or of internal helper class *CentralizedProcess* have been omitted. The latter class is necessary to transform all the processes of a client/server application into threads, such that they can be run in a single-process virtual machine. This is an artifact of the specific (complex) application chosen, and discusses extensively in previous publications [1, 4]. Process centralization makes it possible to select several interacting processes in a single-process model checker. All processes are converted to threads, and network communication using TCP/IP method calls is substituted by a "virtual" network, which connects the centralized processes. For all intents and purposes, the application behaves as if it were a multi-process application, but it runs inside a single wrapper process.

- The formatting has been improved and enhanced with type setting language keywords in bold face.

- Duplicate reports of execution steps referring to the same line of code have been deleted. Such duplication arises, for instance, when one line contains several instructions (such as a string concatenation), or when a lock is acquired, which blocks execution of that thread until the lock is available.

- The thread IDs in the thread stacks (at the bottom of the trace) have been adjusted in order to correspond to the thread IDs inside the error trace. This corresponds to a feature that is available in a newer version of JPF.

Without further explanation, the error trace is very hard to understand, due to its length and complexity. Six threads are involved, five of which execute at least some instructions in this error trace. Three of these threads are still active at the time when the exception occurs. In order to make the error trace more understandable, each step is explained briefly:

- First, the exception of the current thread is shown. This is thread 4 (the last thread shown in the error trace, in transition 116).

- Transition 0 (thread 0): The wrapper thread starts all the processes of the client/process application.

- Transitions 1 – 5 (thread 1): Initialization of the main process of the chat server.

- Transitions 19 – 20 (thread 2): Initialization of the first chat client.

- Transition 22 (thread 2): The first chat client connects to the server.

- Transitions 37 – 39 (thread 1): The chat server accepts the incoming connection and initializes the first worker thread. Note how reference *out* is initialized to `null` in line 19 of `ChatServer.java`. After registration of the new worker thread in the array of active worker threads, the chat server main thread waits again for an incoming connection by calling `servsock.accept` (Transition 39).

- Transitions 56 – 60 (thread 2): The first chat client sends a message to the server and tries to read the first incoming message from the server. This could be its own message or a message from a different chat client.
- Transitions 63 – 65 (thread 3): Initialization of the second chat client, which connects to the server.
- Transitions 77 – 80 (thread 1): The chat server accepts the incoming connection, initializes the second worker thread, and registers that worker thread in the array of active worker threads. After that, the limit of connections has been exhausted, so the main server thread shuts down (transition 80).
- Transitions 81 – 88 (thread 3): The second chat client sends a message to the server and tries to read the server response.
- Transitions 91– 112 (thread 4): The first worker thread initializes its output stream *(out)*, reads the first message from its client, and proceeds to send that message to all active clients using `sendAll`.
- Transition 115 (thread 2): The first chat client reads the server response, closes its connection, and terminates.
- Transition 116 (thread 4): The first worker thread accesses reference *out* of the second worker thread. This reference is `null`. Hence, its access leads to the `NullPointerException` shown at the beginning of this error trace.
- Subsequently, the stack traces of the other active threads are shown. These are thread 3, which is the second client waiting for the response, and thread 5, which is the second worker thread. Method `run` of the second worker thread is eligible for execution, but has not scheduled for execution yet. Hence, reference *out* of the second worker thread has never been initialized.

References

[1] C. Artho and P.-L. Garoche. Accurate centralization for applying model checking on networked applications. In *Proc. 21st Int'l Conf. on Automated Software Engineering (ASE 2006)*, Tokyo, Japan, 2006.

[2] C. Artho, K. Havelund, and A. Biere. High-level data races. *Journal on Software Testing, Verification & Reliability (STVR)*, 13(4):220–227, 2003.

[3] C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient Dynamic Analysis for Java. In *Proc. 16th Int'l Conf. on Computer Aided Verification (CAV 2004)*, volume 3114 of *LNCS*, pages 462–465, Boston, USA, 2004. Springer.

[4] C. Artho, C. Sommer, and S. Honiden. Model checking networked programs in the presence of transmission failures. In *Proc. 1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE 2007)*, Shanghai, China, 2007.

[5] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd Int'l Conf. on Software Engineering (ICSE 2000)*, pages 439–448, Limerick, Ireland, 2000. ACM Press.

[6] L. Cousot and K. Havelund. Visualization of Concurrent Java Program Executions. NASA Ames Research Center, Internal project, 2001.

[7] C. Csallner and Y. Smaragdakis. DSD-Crasher: A hybrid analysis tool for bug finding. In *Proc. Int'l Symposium on Software Testing and Analysis (ISSTA 2006)*, pages 245–254, 2006.

[8] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: Efficiently computing the happens-before relation using locksets. In *Proc. 1st Combined Int'l Workshops on Formal Approaches to Software Testing and Runtime Verification (FATES 2006 and RV 2006)*, volume 4262 of *LNCS*, pages 193–208, Seattle, USA, 2006. Springer.

[9] M. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, 2000.

[10] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proc. 20th IEEE Int'l Parallel & Distributed Processing Symposium (IPDPS 2003)*, page 286, Nice, France, 2003. IEEE Computer Society Press.

[11] The Apache Foundation, 2006. `http://www.apache.org/`.

[12] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL 1997)*, pages 174–186, Paris, France, 1997. ACM Press.

[13] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.

[14] T. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[15] Ø. Haugen. From MSC-2000 to UML 2.0 - the future of sequence diagrams. In *Proc. 10th Int'l SDL Forum Copenhagen on Meeting UML (STL 2001)*, pages 38–51, London, UK, 2001. Springer-Verlag.

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

[17] L. Lamport. How to Make a Multiprocessor that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 9:690–691, 1979.

[18] X. Li, Z. Liu, and J. He. A formal semantics of UML sequence diagrams. In *Proc. 16th Australian Software Engineering Conf. (ASWEC 2004)*, Melbourne, Australia, 2004. IEEE Computer Society.

[19] Microsoft Corporation. *Microsoft Visual C# .NET Language Reference*. Microsoft Press, Redmond, USA, 2002.

[20] O. Radfelder and M. Gogolla. On better understanding UML diagrams through interactive three-dimensional visualization and animation. In *Proc. Int'l Conf. on Advanced Visual Interfaces (AVI 2000)*, pages 292–295. ACM Press, New York, 2000.

[21] J. Roberts and C. Zilles. TraceVis: an execution trace visualization tool. In *Proc. Workshop on Modeling, Benchmarking and Simulation (MoBS 2005)*, Madison, USA, 2005.

[22] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Object Technology Series, 1998.

[23] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[24] M. Smith, G. Holzmann, and K. Etessami. Events and Constraints: a Graphical Editor for Capturing Logic Properties of Programs. In *Proc. 5th IEEE Int'l Symposium on Requirements Engineering (RE 2001)*, August 2001.

[25] B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 1997.

[26] K. Tei. Personal communication, 2006.

[27] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.

[28] J. Yang. Automatically Inferring Temporal Properties. In *Doctoral Symposium, 27th Int'l Conf. on Software Engineering (ICSE 2005)*, St Louis, USA., 2005.

```
java.lang.NullPointerException: calling
        'println(Ljava/lang/String;)' on null object
    at Worker.send(ChatServer.java:44)
    at ChatServer.sendAll(ChatServer.java:91)
    at Worker.run(ChatServer.java:32)
----------------------------------- path to error (length: 117)
Tr. #0 Thread #0
  ChatSim.java:12      Thread t[] = new Thread[nClients + 1];
  ChatSim.java:13      t[0] = new CentralizedProcess(0) {
  ChatSim.java:17      t[0].start();
  ChatSim.java:31      for (int i = 1; i <= nClients; i++) {
  ChatSim.java:32      t[i] = new CentralizedProcess(i) {
  ChatSim.java:37      t[i].start();
  ChatSim.java:31      for (int i = 1; i <= nClients; i++) {
  ChatSim.java:32      t[i] = new CentralizedProcess(i) {
  ChatSim.java:37      t[i].start();
  ChatSim.java:31      for (int i = 1; i <= nClients; i++) {
  ChatSim.java:39      }
Tr. #1 Thread #1
  ChatSim.java:15      new ChatServer(nClientsServed);
  ChatServer.java:51   public ChatServer(int maxServ) {
  ChatServer.java:52   int port = 4444;
  ChatServer.java:53   workers = new Worker[2];
  ChatServer.java:56   ServerSocket servsock = new ServerSocket(port);
Tr. #5 Thread #1
  ChatServer.java:57   while (maxServ-- != 0) {
  ChatServer.java:58   sock = servsock.accept();
Tr. #19 Thread #2
  ChatSim.java:34      ChatClient.main(null);
  ChatClient.java:13   static int currID = 0;
  ChatSim.java:34      ChatClient.main(null);
  ChatClient.java:16   new ChatClient().exec();
  ChatClient.java:19   public ChatClient() {
Tr. #20 Thread #2
  ChatClient.java:20   synchronized(getClass()) {
  ChatClient.java:22   }
Tr. #22 Thread #2
  ChatClient.java:23   }
  ChatClient.java:16   new ChatClient().exec();
  ChatClient.java:27   Socket socket = new Socket();
  ChatClient.java:28   InetSocketAddress addr = new InetSocketAddress(...);
  ChatClient.java:29   socket.connect(addr);
Tr. #37 Thread #1
  ChatServer.java:58   sock = servsock.accept();
Tr. #38 Thread #1
  ChatServer.java:60   synchronized(this) {
  ChatServer.java:61   for (i = 0; i < workers.length; i++) {
  ChatServer.java:62   if (workers[i] == null) {
  ChatServer.java:63   workers[i] = new Worker(i, sock, this);
  ChatServer.java:8    class Worker implements Runnable {
  ChatServer.java:63   workers[i] = new Worker(i, sock, this);
  ChatServer.java:15   public Worker(int n, Socket s, ChatServer cs) {
  ChatServer.java:16   this.n = n;
  ChatServer.java:17   chatServer = cs;
  ChatServer.java:18   sock = s;
  ChatServer.java:19   out = null;
  ChatServer.java:20   in = null;
  ChatServer.java:21   }
  ChatServer.java:63   workers[i] = new Worker(i, sock, this);
  ChatServer.java:64   new Thread(workers[i]).start();
  ChatServer.java:65   break;
  ChatServer.java:67   } if (i == workers.length) {
  ChatServer.java:70   }
```

Figure 10. Part 1 of error trace: Start of server thread, first client, first worker thread.

```
Tr. #39 Thread #1
  ChatServer.java:71   }
  ChatServer.java:57   while (maxServ-- != 0) {
  ChatServer.java:58   sock = servsock.accept();
Tr. #56 Thread #2
  ChatClient.java:30   System.out.println("Client " + id + " connected.");
  ChatClient.java:31   InputStreamReader istr =
  ChatClient.java:33   BufferedReader in = new BufferedReader(istr);
Tr. #57 Thread #2
  ChatClient.java:34   OutputStreamWriter out =
  ChatClient.java:36   out.write(id + ": Hello, world!\n");
Tr. #59 Thread #2
  ChatClient.java:37   out.flush();
  ChatClient.java:38   for (int i = 0; i < 1; i++) {
Tr. #60 Thread #2
  ChatClient.java:39   System.out.println(id + ": Received " + in.readLine());
Tr. #63 Thread #3
  ChatSim.java:34      ChatClient.main(null);
  ChatClient.java:16   new ChatClient().exec();
  ChatClient.java:19   public ChatClient() {
Tr. #64 Thread #3
  ChatClient.java:20   synchronized(getClass()) {
  ChatClient.java:21   id = currID++;
  ChatClient.java:22   }
Tr. #65 Thread #3
  ChatClient.java:23   }
  ChatClient.java:16   new ChatClient().exec();
  ChatClient.java:27   Socket socket = new Socket();
  ChatClient.java:28   InetSocketAddress addr = new InetSocketAddress(...);
  ChatClient.java:29   socket.connect(addr);
Tr. #77 Thread #1
  ChatServer.java:58   sock = servsock.accept();
Tr. #78 Thread #1
  ChatServer.java:60   synchronized(this) {
  ChatServer.java:61   for (i = 0; i < workers.length; i++) {
  ChatServer.java:62   if (workers[i] == null) {
  ChatServer.java:61   for (i = 0; i < workers.length; i++) {
  ChatServer.java:62   if (workers[i] == null) {
  ChatServer.java:63   workers[i] = new Worker(i, sock, this);
  ChatServer.java:15   public Worker(int n, Socket s, ChatServer cs) {
  ChatServer.java:16   this.n = n;
  ChatServer.java:17   chatServer = cs;
  ChatServer.java:18   sock = s;
  ChatServer.java:19   out = null;
  ChatServer.java:20   in = null;
  ChatServer.java:21   }
  ChatServer.java:63   workers[i] = new Worker(i, sock, this);
  ChatServer.java:64   new Thread(workers[i]).start();
  ChatServer.java:65   break;
  ChatServer.java:67   } if (i == workers.length) {
  ChatServer.java:70   }
Tr. #79 Thread #1
  ChatServer.java:71   }
  ChatServer.java:57   while (maxServ-- != 0) {
  ChatServer.java:75   }
Tr. #80 Thread #1
  ChatServer.java:76   System.out.println("Server shutting down.");
  ChatServer.java:77   }
  ChatSim.java:15      new ChatServer(nClientsServed);
  ChatSim.java:16      }};
Tr. #81 Thread #3
  ChatClient.java:30   System.out.println("Client " + id + " connected.");
```

Figure 11. Part 2 of error trace: Second client connects, second worker thread created.

```
Tr. #84 Thread #3
  ChatClient.java:31   InputStreamReader istr =
  ChatClient.java:33   BufferedReader in = new BufferedReader(istr);
Tr. #85 Thread #3
  ChatClient.java:34   OutputStreamWriter out =
  ChatClient.java:36   out.write(id + ": Hello, world!\n");
Tr. #87 Thread #3
  ChatClient.java:37   out.flush();
  ChatClient.java:38   for (int i = 0; i < 1; i++) {
Tr. #88 Thread #3
  ChatClient.java:39   System.out.println(id + ": Received " + in.readLine());
Tr. #91 Thread #4
  ChatServer.java:24   System.out.println(... + Thread.currentThread());
Tr. #93 Thread #4
  ChatServer.java:26   out = new PrintWriter(sock.getOutputStream(), true);
Tr. #97 Thread #4
  ChatServer.java:27   assert(out != null);
Tr. #98 Thread #4
  ChatServer.java:28   in = new BufferedReader(new
Tr. #102 Thread #4
  ChatServer.java:30   String s = null;
  ChatServer.java:31   while ((s = in.readLine()) != null) {
Tr. #105 Thread #4
  ChatServer.java:32   chatServer.sendAll("[" + n + "]" + s);
Tr. #111 Thread #4
  ChatServer.java:89   for (i = 0; i < workers.length; i++) {
Tr. #112 Thread #4
  ChatServer.java:90   if (workers[i] != null)
  ChatServer.java:91   workers[i].send(s);
  ChatServer.java:44   out.println(s);
  ChatServer.java:45   }
  ChatServer.java:89   for (i = 0; i < workers.length; i++) {
  ChatServer.java:90   if (workers[i] != null)
  ChatServer.java:91   workers[i].send(s);
  ChatServer.java:44   out.println(s);
Tr. #115 Thread #2
  ChatClient.java:39   System.out.println(id + ": Received " + in.readLine());
  ChatClient.java:38   for (int i = 0; i < 1; i++) {
  ChatClient.java:41   out.close();
  ChatClient.java:44   }
  ChatClient.java:45   }
  ChatClient.java:17   }
  ChatSim.java:35      }
Tr. #116 Thread #4
  ChatServer.java:44   out.println(s);
------------------------------------ end error path

------------------------------------ thread stacks
Thread #3:
    at java.io.BufferedReader.readLine(java/io/BufferedReader.java:292)
    at java.io.BufferedReader.readLine(java/io/BufferedReader.java:362)
    at ChatClient.exec(ChatClient.java:39)
    at ChatClient.main(ChatClient.java:16)
    at ChatSim$2.run(ChatSim.java:34)

Thread #5:
    at Worker.run(ChatServer.java:24)
------------------------------------ end thread stacks

===================================
  1 Error Found: uncaught exception
===================================
```

Figure 12. Part 3: The second client sends a message, leading to an exception in the server.