Visualization and Debugging of Concurrent Java Programs with UML

Katharina Mehner, Bernd Weymann

Department of Mathematics and Computer Science
University of Paderborn
D-33095 Paderborn, Germany
{mehner|pascal}@upb.de

1 Motivation

Programming and *debugging* are the best understood activities during the development of sequential object-oriented systems, well supported by integrated software development environments consisting of editor, compiler, and standard debugger. The increasing need for distributed applications, e.g. client/server computing over the internet, requires the transition from sequential to *concurrent* systems. Modern languages like *Java* support concurrent programming with threads through the language syntax [2]. While programming languages improve, *debugging facilities fall behind* and have to be extended to cover also errors specific to concurrent programs.

Errors in concurrent systems are more complex than in sequential systems because of the *multiple flows of control* and the inherent *nondeterminism*. Typical errors can only be understood by looking at the execution history. Neither traditional debugging, displaying one system state at a time, nor textual traces from a program run really help to understand such errors. Instead, a *graphical visualization* of the program execution over time is needed. In addition, *automated* support to *detect* and *analyze* errors can ease the developers work.

A key issue is the choice of the graphical visualization. We aim at a better integration of visualization into the software development lifecycle from the language perspective. The de facto standard for visual modeling of object-oriented systems is the *Unified Modeling Language*(UML) [9], a set of languages for describing structure and behavior of software systems at different levels of abstraction. UML provides *extension mechanisms* for adaption to specific problem domains. Using UML, the number of languages used during the development is minimized. More important, automated consistency checks between the design and the visualizations from the program execution are possible. None of the existing visualizations for debugging uses UML. The question is, to what extent UML can be used to visualize concurrent object-oriented program executions and their errors.

A further issue is the relationship between debugging and *testing*. The development of concurrent system requires extensive testing. The execution of test plans is traced [1] and debugging starts from such traces. Therefore, debugging tools either have to integrate testing facilities or provide an import interface for traces. None of the existing visualizations for debugging emphasizes the interoperability. A *standard exchange format* for traces could use UML profiling mechanisms. Testing also requires to handle a large number of traces during debugging and the possibility to compare them. Therefore, a post mortem visualization is primarily needed.

In existing visualizations not all kind of concurrency errors are covered by automated analysis and an appropriate visualization. Therefore, our work is driven by a systematical study of concurrency errors. There are *safety* errors such as data inconsistency and unwanted *race conditions*, and *lifeness* errors such as *deadlocks, lock-outs* or *dormancy*. Errors also occur with high level synchronization strategies like reader/writer-schedules or barriers.

We have developed a *visualization and debugging environment for concurrent Java* based on UML, covering a *general purpose* visualization of the execution of Java programs and the *detection, analysis,* and *visualization* of deadlocks so far. Traces are generated in a non invasive approach, i.e. without changing the source code, based on the Java Platform Debugger Architecture [5]. The post mortem visualization is integrated into the UML-CASE tool Together [12].

2 Visualization of Concurrent and Object-Oriented Programs

Existing visualizations for *concurrent non object-oriented* languages provide high level views on threads with event history, thread state and communication between threads [6, 7, 13]. They only give a first hint to possible errors but then require manual inspection.

There is no visualization with special support for safety errors. It is easy to analyze a trace for data which is accessed simultaneously by multiple threads, but these kind of errors are application specific and therefore require customizable analysis. Race conditions are addressed e.g. by TraceViewer [6]. The trace is analyzed for access conflicts and unordered or concurrent events. This partial order is visualized by a graph. Again, the decision between allowable and unwanted race conditions is application specific and not addressed further on. The visualization from MAD [7] addresses anomalies by a variety of pattern-based analysis, including race conditions.

The primary problems addressed by the GThreads [13] environment are lock contention and deadlocks. Its MutexView (see figure 1) visualizes threads and mutual exclusion locks. A bigger disk symbolizes a lock, the small disks symbolize the threads, distinguished by color. A thread having a lock is inside the disk for the lock, threads waiting for the lock are outside. This view is animated but does not provide analysis of deadlocks. As the access dependency graph is only given implicitly by the positions of the disks deadlocks are never obvious. Messages which trigger the changes are not shown. This makes it difficult to relate a deadlock situation back to code.

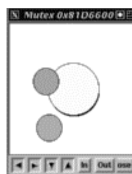

Fig. 1. Mutex View describing locking relations

These visualizations are only partly adequate, because in an object-oriented system it is necessary to visualize the method calls *and* the objects, while in a non object-oriented system the mere function call is sufficient.

Object-oriented visualizations [4, 10, 3] also provide high level views on threads with event history, thread state and communication. In addition, Jinsight [4] from IBM research focus on problems of object-orientation but not on concurrency. The DESERT environment [10] allows to construct user defined visualizations. The visualizations already integrated cover high level analysis or the search for specific patterns in the source code, but don't focus on concurrency errors. Visual Threads [3] focusses on concurrency by supporting POSIX threads and recently also Java threads. This professional tool detects typical errors at runtime, like deadlocks, potential for deadlocks and many more, including customizable analyis. Albeit its strength in the analysis, the visualization of the results is only textual, similar to part of a trace.

Reverse Engineering aims at reconstructing static and dynamic design models from source code and from program executions using UML or similar languages [11]. They do not provide specific analysis and visualizations for errors.

3 Visualizing Concurrent Java Traces with UML

We present a general purpose graphical visualization for the execution of concurrent Java programs and a visualization of deadlocks, based on UML [9]. UML can describe multiple flows of control in structure and behavior diagrams by *active classes* and *objects*, which model the root of a flow of control. Method calls between objects over time can be described with *interaction diagrams*: While *sequence diagrams* have an explicit time axis, *collaboration diagrams* can render additional information about structural relationships and other dependencies. The timing order is presented

by a hierarchical numbering of messages. For the purpose of visualizing program traces UML interaction diagrams are most adequate.

In a previous paper we have discussed in detail the suitability of UML sequence and collaboration diagrams for visualizing deadlocks [8]. Sequence diagrams describe the ordering of messages over time which is important to understand the reasons for a deadlock. In the activation bar of the object lifeline they can show the state of a thread, e.g. blocked. They are the starting point for the presentation of a deadlock. In addition, collaboration diagrams show structural dependencies, but they have to be extended to show the access dependencies involved in the blocking of threads.

A visualization suitable for debugging helps in detecting errors by automated analysis and allows to relate the error back to the code. This requires a fine grained representation, where code statements which are the key to an error are visualized. If language statements are very abstract such as in Java for synchronization, it is difficult to understand the behavior from a visualization of the language statements. An adequate and intuitive *metaphor* helps to visualize the runtime behavior. The Java key word synchronized is used to mark a block of statements or a method which needs exclusive access to the object, on which it is called. To describe the operational meaning of synchronized , we draw on the metaphor of a *lock* for each object. To enter a synchronized -block or method a thread needs to obtain a lock on the object. If multiple threads are trying to obtain such a lock, only one thread is assigned the lock, all other threads are blocked. If a thread leaves a synchronized-region, it releases the lock. We use the extension mechanism of UML *stereotypes* to adapt UML diagrams. A stereotype is a new modeling element which is derived from an existing one. The stereotype name is rendered in guillemets and may have an associated graphical representation. We introduce a new stereotyp for UML-links to describe the meaning of synchronized : either the lock is assigned or the thread is blocked.

We present the visualization language with an example, which is part of a sample banking application, and consists of two Java classes: Account and AccountingTransaction . Account contains the balance of the account in value and its number nr . The method drawValue withdraws money from the account and drawCheque transfers money from another account to this account. Transfers are carried out by the class AccountingTransaction , a Java thread. When it is created, the target and source account and the amount to be transfered are specified. The thread is started with method run .

```
public class Account {
  private long value;
  private int nr;

  Account(int n, long v) {
        nr = n;
        value=0;
        setValue(v);
  }
  public int getNumber() {
        return nr;
  }
  public synchronized long getValue() {
        return value;
  }
  public synchronized void drawValue(
                         long amount) {
        value-=amount;
  }
  public synchronized void drawCheque(
                         Account other,
                         long amount) {
        other.drawValue(amount);
        value+=amount;
  }
}
```

```
public class AccountingTransaction
                          extends Thread {

  Account a1;
  Account a2;
  int amount;

  AccountingTransaction(Account a,
                          Account b,
                          int am) {

        a1 = a;
        a2 = b;
        amount = am;
  }
  public void run() {
        a1.drawCheque(a2,amount);
 }
}
```

We can instantiate a simple scenario with a deadlock: Two threads of class AccountingTransaction transfer money between the same two accounts but in different directions. We demonstrate this situation with a UML sequence diagram (see figure 2). The activation bar on the lifeline of an object is *shaded*, if the object is computing something, it is empty, if the focus of control has left the object or is blocked on another object. The fact, that in the end of the scenario no activation is shaded is the hint to an error. However, the kind of the error can only be determined by further analysis. Here we can easily see, that the two threads are mutually blocked. Note, that this deadlock does not always occur in this scenario as it depends on the order in which the threads call the other objects.

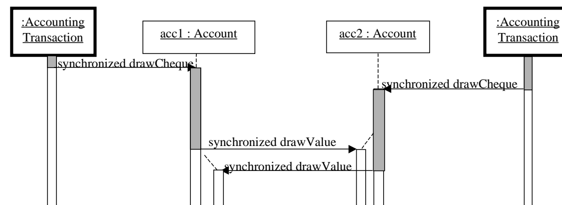

Fig. 2. Deadlock Scenario

We might face this deadlock as part of a bigger program, i.e. the entire sample banking application. The first visualization generated by our tool is a sequence diagram of the execution we have chosen to visualize (see figure 3). On the left is an overview, the dark square being the selection diplayed on the right. For the sequence diagram an equivalent collaboration diagram can be generated by the tool. They both provide an overview of the traces but are not suitable to detect errors.

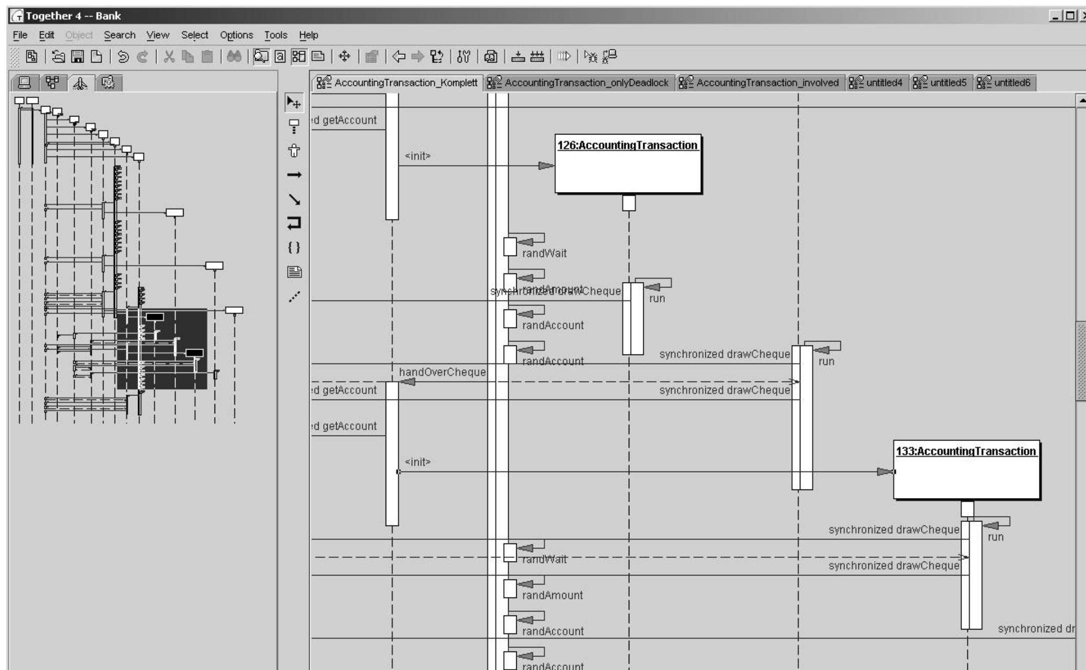

Fig. 3. Generated Sequence Diagram

With a *filter* for *deadlocks*, they are analyzed and each deadlock is visualized with a sequence (see figure 4) and a collaboration diagram (see figure 5). In the sequence diagram detailed infor-

mation about the deadlock is given in a box linked to the threads involved. However, this diagram cannot convey the access dependency graph behind a deadlock.

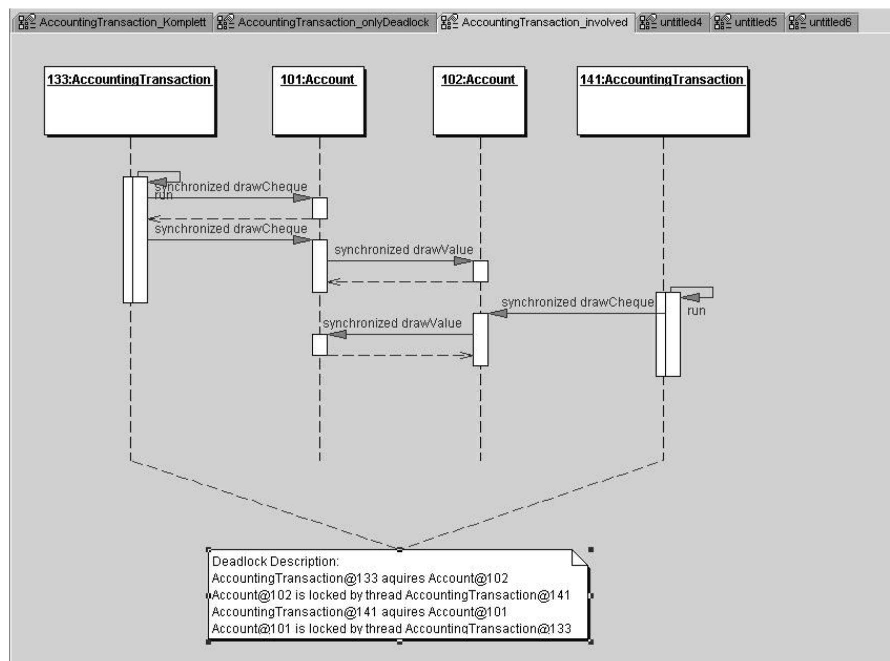

Fig. 4. Deadlock View for Sequence Diagram

The collaboration diagram for the deadlock shows the links between objects involved and is augmented by new stereotype links between objects showing *locking dependencies*. When a synchronized method is called, either the thread gets the lock or is blocked. This is reflected in the new egdes: when the AccountingTransaction -object with the identifier 133 calls 1.2 synchronized drawCheque , it locks it, which is visualized by the directed link from 133:AccountingTransaction to 101:Account labeled with *"locks"*. When it calls 1.2.1 synchronized drawValue , this object is already locked by the other thread and so the thread is blocked, which is visualized by the directed link from 133:AccountingTransaction to 102:Account labeled *"acquires"*. Thereby we can easily understand the cyclic dependencies of the deadlock. (In the tool those edges are red and green and therefore difficult to read on the screenshots.)

4 Implementation

The visualization and debugging environment consists of a *tracing component* for tracing a running Java program, and a *visualization component* for generating UML diagrams from the traces, and allows to analyze the traces for deadlocks and visualizes the results. It is implemented in Java.

The tracing approach is *non invasive* with minimal effect on the execution. It uses Java bytecode and works without starting the program again. This is achieved with the *Java Debugger Architecture* [5] which allows to collect manifold information from a even remotely running Java program, which can be started independently. From this information we generate a trace. For the deadlock detection, we construct an access dependy graph describing the locking relations, with threads and objects as nodes. The graph is updated during tracing. The trace is the starting point for the deadlock detection. Whenever a thread attempts to enter a monitor already locked by another thread, we start a search for a cyclic dependency.

The visualization component translates traces into UML diagrams. We use the UML-Case tool Together [12] to display the UML diagrams. They are imported via the Java interface *OpenAPI* which allows the definition of stereotypes.

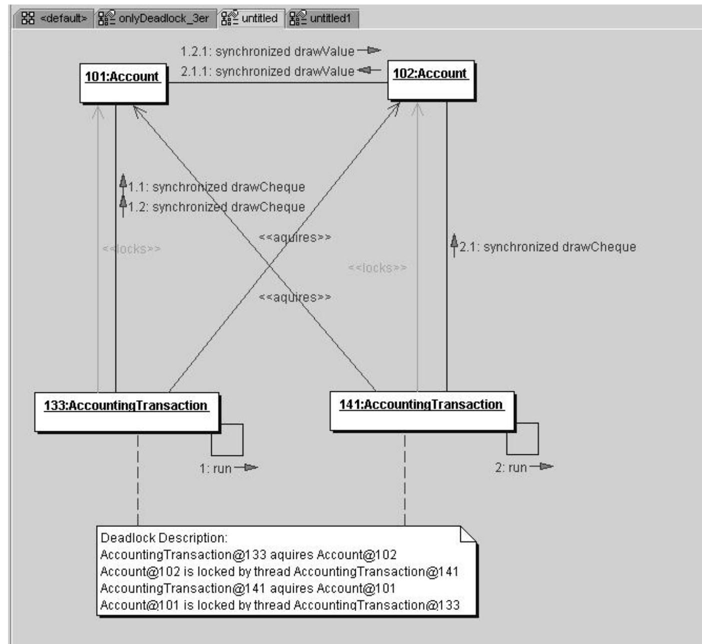

Fig. 5. Deadlock View for Collaboration Diagram

5 Outlook

We will address the analysis of other lifeness errors including high level errors and their presentation in UML in future. Other errors might need static analysis in addition to dynamic analysis. We also want to allow user-defined filters and pattern analysis. We want to put forth the definition of exchange formats for traces to ensure interoperability between tools.

References

1. R. Carver and K. Tai. Replay and testing for concurrent programs. *IEEE Software*, 8(2):66–74, 1991.
2. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1997.
3. J. Harrow. Debugging multithreaded applications on compaq tru64 unix operating systems. Technical report, Compaq Computer, 2000.
4. IBM. Jinsight. http://www.research.ibm.com/jinsight.
5. JavaSoft. Java platform debugger architecture. http://www.javasoft.com/products/jpda/.
6. E. Kraemer. Visualizing concurrent programs. In *Software Visualization: Programming as a Multimedia Experience*, pages 237–256. MIT Press, Cambridge, MA, 1998.
7. D. Kranzmueller, S. Grabner, and J. Volkert. Debugging with the MAD environment. In *Environments and Tools for Parallel Scientific Computing*, volume 23. 1997.
8. K. Mehner and A. Wagner. Visualizing the synchronization of Java-threads with UML. In *IEEE Symposium of Visual Languages*, 2000.
9. Object Management Group. UML specification version 1.3, June 1999. http://www.omg.org.
10. S. Reiss. Software visualization in the desert environment. In *Proc. PASTE*, 1998.
11. T. Systä. On the relationships between static and dynamic models in reverse engineering java software. In *Proceedings of the 6th Working Conference on Reverse Engineering*. IEEE, 1999.
12. TogetherSoft. Together. http://www.togethersoft.com/.
13. Q. Zhao and J. Stasko. Visualizing the execution of threads-based parallel programs. Technical Report GIT-GVU-95-01, Georgia Institute of Technology, Atlanta, GA, January 1995.