Visualizing the Synchronization of Java-Threads with UML

Katharina Mehner Annika Wagner

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

Abstract

Concurrent programming is a complex task, even with modern languages such as Java who provide language-based support for multithreading and synchronization. In addition to typical errors from sequential programming concurrent programming is prone to security and liveness errors, which are difficult to detect due to the inherent nondeterminism in concurrent programs. While debugging is still mainly based on textual representations, we think that the use of visual languages can ease program comprehension. Once a synchronization error is detected, e.g. during testing, the error situation shall be visualized to analyze the reason for the error. With UML being a major visual modeling language for object oriented software development we decide to base our visualization on it and present how to visualize program traces with UML sequence and collaboration diagrams. We focus on the visualization of the synchronization of threads. For this purpose we extend UML to model the runtime mechanisms of the Java language constructs for synchronization.

Keywords: debugging, program comprehension, software visualization, object oriented concurrent programming, Java, UML

1. Introduction

Concurrent programming is becoming more and more important, not only in distributed applications but also in applications for standalone computers. This is taken into account by modern programming languages which support concurrency throughout the language and not only with libraries. An important example is Java [3], which provides high-level language constructs for threads and synchronization. Nevertheless, concurrent programming with Java is still a difficult task. As communication and synchronization is more complex than in sequential systems, in addition to typical errors from sequential programming concurrent programming is prone to security and liveness errors such as deadlocks or dormancy of threads. These errors are difficult to detect due to the inherent nondeterminism in concurrent programs and possible race conditions among threads.

There are several tools for testing concurrent programs. They generate real or fictitious program traces, i.e. protocols of the run of a program, but so called replay tools also enforce the execution of a given trace which allows to circumvent the nondeterminism of concurrent programs for testing purposes. The tool JaDA [1] provides this support for Java. Such traces are usually presented textually.

As textual representations have been found inadequate to express complex error situations as those found in concurrent programming [6] we investigate graphical representations of program traces, which can ease program comprehension. To enable understanding of errors such as deadlocks or dormancy of threads it is not sufficient to visualize program traces on the level of the programming language, as the Java language constructs for synchronization are rather abstract. To be able to analyze errors, it is necessary to show what is hidden behind the language constructs, i.e. from what mechanisms they do abstract. Therefore, it is our aim to visualize the relevant runtime mechanisms behind the Java language constructs for synchronization.

For the visualization we use the Unified Modeling Language [2,9]. The UML is an object oriented modeling language for visualizing, specifying, constructing, and documenting the artifacts of a software intensive systems. UML is widely used in industry and academia and has become an OMG standard, supported by a large number of important companies. UML is a set of visual languages for the modeling of both software systems and business processes. The languages are tailored to specific aspects of a system to be modeled and can be grouped into four categories: use case diagrams, structural diagrams, behavioral diagrams and implementation diagrams. Beyond the stan-
standard modeling elements, UML provides extension mechanisms, e.g. to be adapted to specific problem domains.

Using the same language for the visualization as for the modeling has the advantage that the UML diagrams created from program traces can be compared with those from the design model and checked for consistency. Moreover, there is the possibility to include them in the forward engineering, which is an important prerequisite for Round Trip Engineering. Lastly, the developer doesn’t have to learn another language for the visualization.

For the visualization of program traces we use UML interaction diagrams which are intended to describe scenarios on instance level. While sequence diagrams can capture the behavior over time, collaboration diagrams can render additional information about structural relationships, which are needed to find e.g. the reasons for a deadlock. We use the UML extension mechanisms to adapt collaboration diagrams for modeling the runtime behavior of the Java synchronization mechanisms.

The paper is organized as follows. In section 2 we present the concurrency model of Java. In section 3 we give an overview of UML for modeling concurrent systems. In section 4 and 5 we present how to visualize the most important concepts from the Java concurrency model, mutual exclusion and thread cooperation. In section 6 we sketch our ideas for tool support. We discuss related work in section 7 and give a summary and conclusion in section 8.

2. Concurrent Programming with Java

Java supports concurrent programming with threads through the language and the runtime system [3,6]. A thread is a single sequential flow of control within a program and cannot run on its own. Java handles threads in an object oriented way and provides a thread class library that defines a set of operations on threads.

Threads within the same program can access the same objects to implement their functionality. But also communication between threads is via shared objects. To avoid inconsistent states of the commonly used objects the access on these objects must be synchronized. Moreover, threads have to cooperate to exchange information, i.e. they sometimes must wait for each other.

Java supports synchronization through the use of monitors. A monitor provides mutual exclusive access to data, i.e. only one thread at a time has access to this data. The monitor also supports cooperation between threads. It has mechanisms that can be used by threads to wait for a certain state of the data or to signal that the data has changed. A monitor keeps the waiting threads in an associated wait-queue.

Any object in Java can serve as a monitor: The keyword synchronized is used to mark a region in the code which needs exclusive access to the object, also called a critical region. With the methods wait, notify and notifyAll of a monitor the wait-queue of the monitor (i.e. the object) can be manipulated.

A synchronized-region always grants exclusive access to an entire object. If synchronized is applied to a method it is the object on which the method is called. If it is applied to a block of statements, the corresponding object has to be specified. To be able 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 locked out and they are blocked, i.e. they cannot proceed. If a thread leaves a synchronized-region, it releases the lock. By non-deterministic choice one of the blocked threads will receive the lock and can start to execute on that object. In the blocking of threads lies the potential for a deadlock. The easiest case consists of two threads each of which has locked an object and then tries to obtain the lock on the other object. Thus the necessary condition for a deadlock is, that there is a cyclic relation between the objects used by threads. Deadlocks are difficult to detect. When it comes to a deadlock, the reason is often obscure. The context and the execution history is needed to analyze the deadlock. Due to the nondeterminism in concurrent programs deadlocks need not occur, even if programs have the potential for a deadlock.

Wait and notification methods can only be used within a synchronized-region. When wait is called, the thread changes its state from running to waiting and releases its lock. Then it is inserted into the wait-queue. Threads can only be removed from this queue by notifications on the same object. Using notify removes only one thread from the queue while notifyAll removes all threads from the queue. Such a notification can only be issued by an active thread. If there is no such notification or not even an active thread, then the waiting threads can never become running again. This situation is known as the dormancy problem. A dormant thread is a thread that fails to become running again. Therefore it is important that wait methods be balanced by notification methods.

3. Modeling concurrent programs with UML

UML is well suited for modeling concurrent systems. It can describe multiple flows of control in structure diagrams as well as in behavior diagrams. UML provides the notion of active classes and active objects to model the root of a flow of control. When an active object is created, the associated flow of control is started. When it is destroyed, the flow is terminated. An active object owns either a process or a thread. A process is a heavyweight flow of control which has its own address space.
is a lightweight flow of control which runs concurrently with other threads within a process and shares its address space with them. Communication between threads or processes can be described with interaction diagrams. Thus for the purpose of visualizing program traces we will use UML interaction diagrams. In the setting of the paper we are not interested in class diagrams. If our visualization techniques are used in an object-oriented development process, the class diagrams are already given with the design model and it is not necessary to infer them from the program.

In order to be able to describe Java programs at runtime we will use one of the UML’s extension mechanisms: stereotypes. Stereotypes extend the syntax of UML. They create 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 which can be used instead of the stereotype name.

In the following we aim at representing the monitor concept, which we do in two steps. First we discuss how to visualize the mutual exclusion. Next we consider the visualization of thread coordination with wait and notification methods.

4. Visualizing mutual exclusion

In this section we focus on the visualization of mutual exclusion. We will see that two different aspects are important for the visualization which can be expressed by two different kind of UML diagrams.

4.1. Sequence diagrams

To understand the effects of synchronization it is essential to visualize the exact time ordering of messages sent in a concurrent system. Different time orderings can yield different results. There is only one UML diagram which can capture traces in their time ordering: the sequence diagram.

```
class cell {
    private int value;
    public void swap(cell other){
        synchronized(this){
            synchronized(other){
                int newValue = other.value;
                other.value = value;
                value = newValue;
            }
        }
    }
}
```

Our running example will be the code of class `cell` (figure 1). A cell has a content `value` and a method `swap` to exchange its value with another cell. The exchange must be atomic, i.e. during that time no other objects are allowed to read or write the contents of those cells. To ensure this, the method `swap` first acquires all necessary locks via `synchronized`, i.e. the locks for the two objects involved in the exchange (like in a two-phase-lock protocol) before it actually swaps the contents.

The simplest situation where objects of class `cell` are used is the one, where two cells are requested independently by two different threads to exchange their contents. Due to nondeterministic progress of threads this can result in different traces, two of which are visualized in the following sequence diagrams (see figure 2 and 3). Note, that they are based on the same Java program. We denote the two threads as active objects `client1` and `client2`. Note, that the shading of the focus of control or activation in the sequence diagram represents, that something is computed, while the empty activation indicates, that the object has to wait for a message to return. (This is a new notation from UML 1.3). If none of the above applies, the lifeline is dashed.

Apart from the method calls we have to visualize synchronized-blocks. Without doing so, the sequence diagram wouldn’t render the expected information about synchronization. To represent a synchronized-block we look at its semantics. At the beginning a lock is acquired for the object given as the parameter. The thread has to wait until it receives the lock. We therefore denote the beginning of the block as a synchronous method call to the object, of which the lock is acquired. Analogously, at the end of the block the lock is returned via a synchronous call to the object.

Figure 2. Sequence diagram
The first diagram shows a successful exchange: cell1 can acquire all locks needed. Then cell1 swaps its content with cell2, which is denoted with ... Then cell1 releases all its locks. Cell2 acts analogously.

The next diagram shows a deadlock (figure 3). This can be deduced from the fact, that no activation is shaded at the end of the interaction, i.e. they are all waiting for something in order to be able to continue. Moreover, the diagram shows the execution order that led to a deadlock. While cell1 tries to exchange its contents with cell2, cell2 already tries to do it vice versa. However, the diagram does not show, why this order leads into a deadlock. The possibility for deadlocks is given with the semantics of synchronized: objects acquiring a lock are blocked until they will receive it. The fact that another object can already possess the lock is a special kind of relationship between the locking object and the locked object. Relations cannot be made visible in a sequence diagram. Therefore we continue our deadlock analysis with collaboration diagrams, which can visualize relationships.

4.2. Collaboration diagrams

In this section we show how collaboration diagrams together with appropriate extensions can help to analyze the reasons for a deadlock. We can translate the second sequence diagram from figure 3 into a collaboration diagram (figure 4). Again we denote the threads as active objects. We show the same calls as before, including the interpretation of a synchronized-block. We have to translate the time ordering into a sequence numbering for each thread separately. The sequence numbers of concurrent threads are preceded by different capital letters, here A and B. Sub-calls are numbered using nesting and comprise also the synchronized-blocks. This results in a time ordering only within one thread, but we lose the time ordering in which shared objects are visited by different threads. For this purpose collaboration diagrams have means to describe timing constraints, e.g. we can state that B.1 takes place before A.2 and A.2 takes place before B.2. The links between the two cells exist only based on parameter passing, therefore the link end is marked with the stereotype <<parameter>>. The constraint [new] specifies, that the link is created within the collaboration.

Note that without timing constraints the collaboration diagram for the first sequence diagram from figure 2 would look the same. Moreover, the collaboration diagram in its present form does not even show the occurrence of a deadlock, because it cannot describe the state of an activation, though our aim was to show the internal structure of a deadlock.

4.2.1. Stereotyped links. As we said before, possessing a lock can be seen as a relation between the locking object and the locked object, which can be denoted as a directed link in our collaboration. However such link has no corresponding association in a class diagram and it can’t be navigated either. As this is a special kind of link we introduce a new stereotype <<locks>>. The situation for an object which tries to acquire a lock is handled analogously. Here we introduce the new stereotype <<acquires>>.

Now we can add the missing information in the collaboration diagram (figure 5). Client1 has the lock for cell1 after the call A.1: synchronized(this), which is a directed link of stereotype <<locks>>. Within the call A.2: synchronized (cell2)
the client1 is blocked which we denote with a directed link from client1 to cell2 of stereotype <<acquires>>. We mark the additional links as {new} as they are not destroyed during the collaboration if it comes to the deadlock. Now we can analyze the reasons for the deadlock. We see, that active objects with an <<acquires>>-link are blocked. Looking for cyclic dependencies we can see, that the threads block each other. To completely analyze the deadlock we need to assign the links to their corresponding calls which created them.

In this example it is easy, as the synchronized-calls are on different objects and in the hierarchy directly under swap. It is impossible to assign those links to synchronized blocks if there are two blocks one after another for the same object. It is also impossible to assign those links if there are nested method calls to an object and the synchronized block is somewhere in that hierarchy and you want to know on what level of nesting the <<locks>> or <<acquires>>-links were created. The solution is presented in the next section.

4.2.2. Sequence numbers for links. The assignment of <<locks>> or <<acquires>>-links to the method calls or synchronization-statements who created them and to the level in the hierarchy at which they were created is needed to localize the pieces of code which are the reasons for a deadlock. This assignment can be achieved, if we incorporate links into the sequence numbering of method calls. A newly created link is treated like a method call and has its own number. In the example the call A.2 synchronized(cell2) first leads to the creation of an <<acquires>>-link. This is a sub-action within the synchronized-block and thus gets the sequence number A.2.1.

![Image: Collaboration with numbered links](image_url)

**Figure 6. Collaboration with numbered links**

Normally, links in UML collaboration diagrams are not numbered because these diagrams abstract from the concrete order in which they are created. In our case we need to show them. The abstraction does not make sense for our purpose of visualizing concrete program traces.

4.2.3. Temporary active objects. In the previous sections active objects played a central role in visualizing deadlocks. They are the ones who possess locks and who wait for locks, denoted by corresponding links. However in a concrete deadlock situation they may not be of interest. To the contrary, having to include them in the sequence and collaboration diagrams can make the diagrams more complex than it is needed. In this section we show, how we can reduce complexity by considering only the relevant parts of an interaction. We can replace active objects by introducing the stereotype of temporary multiply active objects.

Active objects have their own focus of control in contrast to passive objects. During execution this focus leaves the active objects when using other objects. When a method is called on another object, the focus moves to this object. The object being called becomes temporarily active, as it has the focus of control for a certain time.

The relevant part of a collaboration diagram will always contain the objects being called. To represent the threads being involved in the interaction it is sufficient to mark the objects being called as temporary active using a stereotype. For this stereotype we choose a graphical representation. Similar to the representation of active objects we draw a thick border line, but it is dashed to denote that it is only temporary.

![Image: Collaboration diagram with temporary active objects](image_url)

**Figure 7. Collaboration diagram with temporary active objects**

To show that objects are locked by threads we assign the <<locks>>- and <<acquires>>-links now to the temporary active objects. Of course, all objects in a collaboration diagram are temporary active, as they are all involved one or more times in the interaction. It does not make sense to mark them all as temporary active objects. Therefore we mark only the last temporary active object within a thread as such. This does not render new information, as this can be inferred from the sequence numbers as well. However, the visualization does render this information in a direct fashion.

The collaboration in figure 7 presents the same trace as before, only the active objects client1 and client2 have been...
4.2.4. Temporary multiply active objects. In a multi-threaded context there can be several threads working in an object at the same time, if it is not mutual exclusive. Threads can execute either different methods or the same method. This makes sense e.g. in the case of multiple readers. To denote an object which is used by several threads simultaneously we introduce the stereotype of a temporary multiply active object. Its graphical representation is similar to the temporary active object, however each additional focus of control is denoted by an additional frame with a thick dashed borderline, which is stacked behind the previous one. The <<locks>> and <<acquires>>-links assigned to the different threads are drawn between the corresponding frame and the locked or acquired objects.

Figure 8. Temporary multiply active object

This is demonstrated with the example in figure 8. Here we assume, that the method swap was called by two different threads on cell1, though with different parameters cell2 and cell3. After these calls there are two foci of control inside the object cell1. The second focus is denoted by the second dashed frame. Only one of them can get the synchronized-lock. In this case it is the thread marked with A, which gets the lock. The call within thread B to synchronized (this) results in an <<acquires>>-link, which is drawn from the second frame to the object cell1.

4.2.5. Complex traces. For demonstration purpose we used a very simple example for a deadlock throughout this section. The potential of a deadlock in the code of class cell is easy to notice for an experienced programmer. Moreover the deadlock can be avoided by applying the resource ordering strategy [5]. In more realistic examples, deadlocks are not local to one class or even one method but they involve several classes and method calls. Then it is even more important to have a visual representation of the trace which lead to a deadlock showing the classes and methods involved.

We can deal with more complex situations, where several synchronized-statements are spread over a sequence of nested methods calls, all on the same object. Then the visualization should not show all the method calls, but only the synchronization-statements together with its directly enclosing method call. This requires to filter traces, i.e. to remove all calls irrelevant for the synchronization.

5. Visualizing thread coordination

In this section we show how sequence diagrams can be used to show errors in the use of wait and notification methods. We look at the example in figure 9. The class buffer represents a passive objects. The buffer contains at maximum one element. This element can be extracted via the method get and a new element can be inserted in the buffer with method put. Mutual exclusion is guaranteed by marking these two methods with synchronized.

```java
class buffer {
    private int item;
    private boolean FULL = false;
    public synchronized int get () {
        while ( !FULL ) {
            try { wait() }
            catch(InterruptedException e) {}  
        }
        FULL = false;
        notifyAll();
        return item;
    }
    public synchronized int put (int i) {
        while ( FULL ) {
            try { wait() }
            catch(InterruptedException e) {}  
        }
        item = i,
        FULL = true;
    }
}
```

Figure 9. Java class buffer

In our example we assume that the buffer is used to exchange data between a producer and a consumer. Here we don’t show the code for the producer and the consumer. The producer is a thread who makes calls to put from time to time and the consumer calls get from time to time. The consumer can only take the contents if there is one. If the buffer is empty, the thread has to wait until the producer
has inserted a new element. This behavior is implemented in method get by a while loop. Unless the condition is true, the thread using the buffer is set to a waiting state. However, if the consumer has successfully extracted the contents it sends a notification, which notifies the waiting producer, that it can now try to insert a new element. The code of put works similarly, however we have omitted the notification. We will see soon, that this can lead to a dormancy problem.

In the following we show two example traces of a producer and a consumer using the buffer. Both use the same code. In both traces the buffer is initially empty. The first sequence diagram in figure 10 shows a trace that is successful. The producer puts an item in the empty buffer. Its second attempt to put a new item only takes place after the consumer has extracted the item. In this way they could continue forever.

Dormancy problems are different from deadlocks. While in deadlock situations we need exact information about the locking relations, here we have no such cyclic dependencies. The case could even be more general, that the threads are not in the same wait-queue but are waiting at different objects. The only information available is, who is waiting where. The sequence diagram is sufficient to render this information and a collaboration diagram would not render additional information. At this point we can only deduce, that notifications are missing, but it is not easy to say, where they are missing in the code.

There are numerous solutions to correct dormancy errors, in this case we know, that the producer has to call notify or even notifyAll if we assume several producers and consumers in our system.

The next sequence diagram shows a situation where all threads become dormant (see figure 11). In this trace, the consumer makes an attempt to extract the item. As the buffer is initially empty the consumer calls wait and releases its lock. The producer who was blocked as it was already trying to acquire the lock can continue. When it tries a second time to insert an item, the buffer is full and the producer also calls wait. Now both threads are in the wait-queue of the buffer. This sequence diagram clearly shows, that there is no thread which can issue a notification.

The next sequence diagram shows a situation where all threads become dormant (see figure 11). In this trace, the consumer makes an attempt to extract the item. As the buffer is initially empty the consumer calls wait and releases its lock. The producer who was blocked as it was already trying to acquire the lock can continue. When it tries a second time to insert an item, the buffer is full and the producer also calls wait. Now both threads are in the wait-queue of the buffer. This sequence diagram clearly shows, that there is no thread which can issue a notification.

Figure 10. Sequence diagram for producer-consumer example

The next sequence diagram shows a situation where all threads become dormant (see figure 11). In this trace, the consumer makes an attempt to extract the item. As the buffer is initially empty the consumer calls wait and releases its lock. The producer who was blocked as it was already trying to acquire the lock can continue. When it tries a second time to insert an item, the buffer is full and the producer also calls wait. Now both threads are in the wait-queue of the buffer. This sequence diagram clearly shows, that there is no thread which can issue a notification.

Figure 11. Sequence diagram with dormant threads

Dormancy problems are different from deadlocks. While in deadlock situations we need exact information about the locking relations, here we have no such cyclic dependencies. The case could even be more general, that the threads are not in the same wait-queue but are waiting at different objects. The only information available is, who is waiting where. The sequence diagram is sufficient to render this information and a collaboration diagram would not render additional information. At this point we can only deduce, that notifications are missing, but it is not easy to say, where they are missing in the code.

There are numerous solutions to correct dormancy errors, in this case we know, that the producer has to call notify or even notifyAll if we assume several producers and consumers in our system.

The means needed to analyze dormancy problems go beyond the power of single program traces. We have to analyze, which conditions lead to the waiting of threads, when do conditions change, which threads have to be informed about changes, and which threads can issue notifications when changes take place. Thus we have to analyze the state of objects and their changes. This requires static analysis such as program slicing (see also section 7) which we can use to search for statements which have side effects on the state. To visualize state and state change we intend to use UML-statecharts. This is still ongoing work.

6. Tool Support

We demonstrated our visualization techniques with simple examples, where it was possible to construct the UML diagrams by hand, which is impossible for complex programs. This should be supported by a tool. Our focus will be on a component for visualizing traces, but not on the collection of traces, which can be done by existing tools, e.g. [1].

Our component should have an interface to use traces from existing tools as those mentioned in the introduction. So far, there is no standard data format for Java program traces, but this is desirable and the exchange of traces could be based on XML. We intend to work on the development of such a format and to keep our tool independent from proprietary trace formats to be able to exchange trace generators. For the meantime we either use a transformation to translate proprietary trace formats into our format or build a simple prototype trace generator. Our visualization component will translate the traces to UML, augmenting it with the synchronization mechanisms.

The presentation should be carried out by UML tools which are already used by the developers to create their analysis and design models such as Rational Rose [7]. The Rose diagram editors can handle UML extensions and al-
low to provide new graphical icons for self defined stereotypes, which is essential for our visualization concept. We think that it is extremely important to integrate the visualization techniques with the existing development processes and case tools to unburden the developer to learn too many tools and formalisms.

7. Related work

Our aim to visualize program traces to detect errors is related to the area of debugging. While usual debuggers allow to inspect the state of a system only or have advanced features for the textual representation of execution history with traces, the advantage of our trace representation is in the use of graphical languages. Moreover, we use two different kind of diagrams which emphasize on different aspects of traces. While sequence diagrams can represent the time ordering of different threads, collaboration diagrams can represent structural relationships. Our approach defers from debugging as we also visualize the runtime mechanisms. The already mentioned testing tool JaDA [1] transforms programs so that they generate traces of synchronization events in textual format. However they do not focus on error situations.

Our approach is related to program comprehension. Static program comprehension is based on the analysis of source code. One of the analysis techniques is program slicing, which means that all statements are computed which have side effects on a given variable. This is adapted to concurrent Java programs in [10].

Dynamic program comprehension uses additional information from program traces. There are already approaches which use graphical formalisms. For instance the tool Jinsight [4] uses message sequence charts. However, they do not cover synchronization or analysis of errors in the way we do.

Dynamic reverse engineering uses the information gathered from multiple traces to infer more general knowledge about the behavior of a program, e.g. using state charts as in [8]. It is important for reverse engineering to abstract from certain details and focus on specific aspects. However this approach does not explicitly cover synchronization behavior.

We view our approach to visualize synchronization complementary to formal approaches of deadlock detection as presented in [11]. Our approach is applicable for deadlock problems where the developers do not want to use formal techniques and for those situations where the formal techniques are limited.

8. Conclusion

We presented an approach to support developers of concurrent programs to analyze errors detected e.g. during testing by the use of a visual language, which is based on UML, as this is a language widely used for object oriented software development. In addition to normal traces we focussed on the visualization of Java runtime mechanisms. UML was extended by stereotypes to add Java specific modeling elements. We showed how UML sequence diagrams can visualize traces and especially visualize liveness errors. We used collaboration diagrams to render additional information about relations which are e.g. necessary to find the reasons for a deadlock. There are two alternative ways how to use collaboration diagrams. Either the threads involved are represented themselves as active objects or it is abstracted from them by using temporary active objects. While collaboration diagrams render additional information for deadlock analysis they are not useful for visualizing dormancy problems. Here we proposed the use of state charts, but this is still ongoing work how to improve the visualization of dormancy problems.

We sketched the first ideas for tool support for our language. For more realistic examples we also have to think about the limits of static representation of dynamic information. Both sequence and collaboration diagrams render dynamic information in a static fashion. While sequence diagrams use the second dimension to map the time ordering, this cannot be used in collaboration diagrams as they use it already for structural information. To be able to render also timing information (within one thread), they have to make use of textual presentations (sequence numbers). Another possibility would be to deliver the same information by animated traces.

We made positive experiences with our language in teaching. It helps students who learn concurrent programming and carry out small scale projects to detect and analyze their errors, because it gives them a graphical means to master the complexity when making the transition from sequential to concurrent programming and can serve as a base for discussions.

Literatur


