Modeling Java Threads in UML

2-97

Martin Schader, Axel Korthaus

Lehrstuhl für Wirtschaftsinformatik III
Universität Mannheim, Schloß
D-68131 Mannheim, Germany

Tel.: +49 621 292 5075
email: {mscha | korthaus}@wifo.uni-mannheim.de

Abstract:

Modern programming languages such as Java facilitate the use of concurrent threads of control within computer applications. Considering concurrency is an important prerequisite for the adequate realization of object-oriented ideas, since real world objects usually operate in parallel. This brings up the requirement to be able to express concurrency issues already in design models, which precede implementation activities. The paper examines the possibilities of modeling Java threads using the Unified Modeling Language (UML). UML is the official industry standard for object-oriented modeling as defined by the Object Management Group (OMG).

Key words:

Unified Modeling Language, Java, Threads, Concurrency, Synchronization, Static Structure Diagrams, Dynamic Diagrams

1 Standardization of the UML

In order to cope with the complexities of modern software products, today's software development methodologies propagate a model-based approach. Graphical models of the system's structure and behavior are built and serve as a high-level, easy-to-understand means of communicating requirements and analysis/design concepts to software developers, management, end users, and customers.

In the field of object technology there are at least 80 different analysis and design methodologies which define notations for building software models and describe the development processes to follow. Most of these methodologies build on the same basic concepts, such as object, class, message, relationship etc., but use slightly different notational elements to visualize them. In addition, the development processes recommended by different methodologies differ considerably.

In June 1996, the Object Management Group (OMG), an industry standards organization, who had recognized the need for a standard, issued a Request for Proposal (RFP) for an Object Analysis and Design Facility in order to define an extensible core set of modeling concepts and to facilitate the interchange of object-oriented analysis and design models between CASE tools. As a by-product this RFP led to the submission of proposals for a standardized object analysis and design modeling language including notations for visualizing the basic concepts. While at first there had been a number of competing proposals, in March 1997 the submitters decided to combine and merge their current proposals and contribute jointly to the further development of the Unified Modeling Language (UML), which was originally worked out by Grady Booch, Jim Rumbaugh and Ivar Jacobson of Rational Software Corp. This effort culminated in the jointly produced UML 1.1 response to the RFP, whose adoption by the OMG as the official industry standard for software design notations was announced on November 17, 1997.

The UML does not prescribe a process and thus is no methodology, but just a modeling language. It is often referred to as a general-purpose "language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems" (Rational Software Corp. (1997c)). Its proponents claim that the UML is well-suited to address current software technology problems such as component technology, patterns, frameworks, distribution, and last but not least concurrency.

2 Modeling Concurrency

The set of basic concepts supported by OO-methods is influenced by the capabilities of the programming languages used for implementing software systems and the operating systems on which the software is executed. The growing popularity of Java, a language which enables and encourages the use of interfaces and threads, has led to the integration of these concepts in analysis and design methodologies formerly lacking support of such concepts (Coad and Mayfield (1997)). In particular, the modeling of concurrency is an aspect often neglected by current object-oriented analysis and design methods, but it becomes more and more important as operating systems support multitasking/-threading and programming languages such as Java integrate language elements for writing code to implement multiple threads of control.

Concurrency denotes the occurrence of two or more activities during the same time interval. In computer-based systems, concurrency can be achieved by interleaving (i.e., concurrency is simulated) or simultaneously executing two or more threads of control, which are single execution paths through a program (Rational Software Corp. (1997b)). In the area of object-oriented analysis, most of the methodologies found in literature support either none (e.g. OOA, Coad and Yourdon (1991)) or only very few rudimentary mechanisms of expressing concurrency issues within analysis models (e.g. MAOOAM, Schader and Rundshagen (1996)). Only a very small number of specialized methods, such as MORE/RT (Stein (1995)), integrate concepts of concurrency. In object-oriented design methods such concepts can be found more frequently, e.g. in Booch (1994), ROOM (Real-Time Object-Oriented Modeling, ObjecTime (1997)), and CODARTS (Concurrent Design Approach for Real-Time Systems, Gomaa (1993)). Allowing for the latest requirements of object-oriented software development and profiting from the experience with existing methods, the UML as a language for building analysis and design models integrates the concepts of concurrency, as we will show in section 3.

In concurrent object-oriented systems, a distinction between *active* and *passive objects* is made. Active objects (belonging to active classes) model the concurrent behavior of real world objects and serve for improving the efficient use of system resources. Active objects own an execution thread and can initiate control activities. Passive objects (belonging to passive classes) on the other hand generally wait for messages from other objects to arrive in order to perform an operation, and execute only within the thread of an active object (or the main program thread) (Rational Software Corp. (1997b), Eriksson and Penker (1998)). Active classes can be implemented by heavyweight processes with their own address space or by lightweight threads sharing the same address space. We will focus on Java threads in this paper, so the latter interpretation is relevant to us here.

Programming with concurrent execution paths is a difficult task. As Booch (1994) puts it: “Indeed, building a large piece of software is hard enough; designing one that encompasses multiple threads of control is much harder because one must worry about such issues as deadlocks, livelock, starvation, mutual exclusion, and race conditions.” A concurrent system is a system where different components execute independently of one another, but often have to communicate with each other and therefore have to synchronize (Stein (1995)). Communication and synchronization are the most important issues to be considered when designing a concurrent system, so the minimum requirement for a modeling language to support the design of concurrent systems is to provide some means of expressing different kinds of communication among active and passive objects as well as their synchronization. Before we will go into detail describing how UML supports this, it should be explained further what is meant by communication and synchronization.

In order to communicate, two objects exchange messages across a link. In programs that contain only one thread of control, message passing leads to simple method calls: control flow is sequential and the caller waits for the supplier to complete the invoked method. In a concurrent system, a sender might for example be able to send an asynchronous message, which means, that it does not have to wait for the receiver to finish the execution of the method called, but may go on with its own execution immediately after sending the message. (Different kinds of communication supported by the UML are introduced later.) Synchronization is a mechanism needed to avoid that different active objects acting simultaneously on the same object interfere with its state and may corrupt it in the worst case. If there are no means of ensuring that

some resources can only be used by one thread at a time, inconsistent system states can be the result. Booch (1994) describes the purpose of synchronization by saying: “In the presence of concurrency, it is not enough simply to define the methods of an object; we must also make certain that the semantics of these methods are preserved in the presence of multiple threads of control.”

The modeling elements for expressing concurrency issues provided by the UML will be introduced in the remainder of this paper, focusing on examples of multi-threaded code snippets in Java. Modeling a system with the UML is done by modeling different views of the system. Therefore, the UML contains a number of different diagram types, all of which serve a certain purpose and emphasize different aspects of the system. The UML’s set of diagrams consists of use case diagrams, static structure diagrams (class and object diagrams), dynamic diagrams (state diagrams, activity diagrams, and interaction diagrams, i.e. sequence diagrams and collaboration diagrams), and implementation diagrams (component diagrams and deployment diagrams) (Rational Software Corp. (1997a)). As can be inferred from the names of the diagrams, there is no special diagram for expressing concurrency. Each diagram may contain information relevant for the modeling of parallel activity, but especially the dynamic diagrams have expressiveness with respect to concurrency.

3 Modeling Java Threads in UML

The exponential growth of the internet, the development of the World Wide Web, and the features of the programming language itself (portability, robustness, security, object-orientation, dynamics, ease-of-use, etc.) have made Java[1] very popular at present. Two of the main advantages of Java are that it has been designed from scratch based on experience gained with existing languages, and that it takes advantage of the features of modern operating systems (e.g. preemptive multitasking/-threading). While many programming languages don’t support concurrency at all or require the use of separate packages or libraries which are hard to use, Java integrates multithreading mechanisms directly into the language. The use of threads in Java still requires the use of classes provided by the standard Java package, but thread synchronization is included at the language level.

In Java, concurrency is introduced through the use of class Thread of package java.lang. Thread extends Object, which is the root class of the Java class hierarchy (see Figure 2). In Java, threads of control can be created either by declaring a class to be a subclass of class Thread and overriding the run method, or by declaring a class that implements the interface Runnable, which means to implement the run method and to pass an instance of that class as a parameter to a new Thread. The use of the second option is compulsary, if the user class is already a subclass of some other class, because Java does not support multiple inheritance. For the purpose of this paper, there are no important differences between the two possibilities, so we will only declare subclasses of Thread.

[1] For a Java language reference, see Sun Microsystems, Inc. (1997); introductory descriptions are provided, for example, by Boon (1996 and 1997) and Ritchey (1995); for concurrent programming with Java, see, for example, Lea (1997).

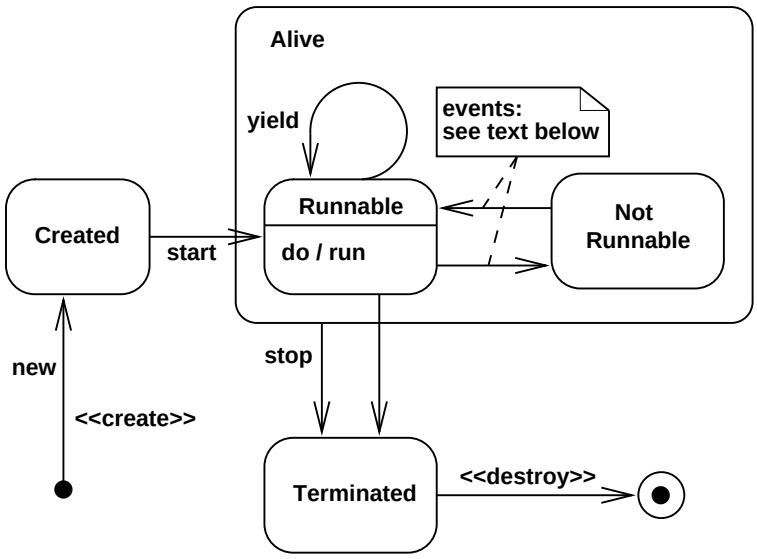

Figure 1: Simplified statechart of Java class Thread

In order to introduce the semantics of Java threads, we fall back on statechart diagrams provided by the UML as a means of modeling the lifecycles of objects in a certain class. These lifecycles are visualized in the form of state machines which show the different states an object may go through during its lifetime and the transitions between those states which model the reaction to received stimuli. Figure 1 shows a simplified statechart, which depicts the lifecycle of objects of class Thread in Java. It shows the thread lifecycle states Created, Alive (with substates Runnable and Not Runnable), and Terminated. It can be seen how composite states may be composed of sequential substates in UML statechart diagrams. Composite state Alive, which is refined by substates Runnable and Not Runnable, is meant to indicate that the thread will respond to the message isAlive() by returning true, and that the thread may be terminated by the stop event at any time, no matter in which substate it is.

Threads executing statements on a CPU are called active. Immediately after its creation a Thread object is not active; first, its operation start() has to be invoked. The object then shifts to state Runnable, which means that it can become active at any time, provided a CPU is available. The note symbol in Figure 1 (the rectangle with a bent corner) indicates that there are several possible events leading to transitions between the two states Runnable and Not Runnable, which are left out in the diagram for the sake of simplicity. Examples of events that may cause a runnable thread to become Not Runnable are that the object calls sleep for itself, that its method suspend is invoked, that it sends a join message to another thread or that method wait is called. A thread in state Not Runnable cannot become active, i.e. cannot execute statements. But, analogously, there are a number of events which may cause the transition of the thread object back into state Runnable. For example, if the period of time specified as an argument of sleep is over, if method resume is called, if the thread who was sent a join has terminated, or if notify or notifyAll is being called. By calling yield a thread indicates that it is ready to become inactive while keeping its state (i.e. Runnable).

The string 'do' '/' machine-name (argument-list) in the lower compartment of a state symbol indicates, that the object performs an activity while in the current state, which can be represented by

a nested state machine. The activity a thread performs is defined in its method run. Completion of the activity triggers the unlabeled transition leaving the state, i.e. completion of run results in the normal termination of the thread. The initial state (shown as a small solid filled circle) and the final state (shown as a bull's eye) are called pseudostates, because objects cannot be in such a state. Transitions from the initial state and to the final state may be labeled with stereotypes ≪create≫ and ≪destroy≫. Stereotypes are an extensibility mechanism of the UML, which will be explained below.

Every Java class with thread functionality owns the above state machine, but may provide additional states, depending on its structure. UML statecharts offer some advanced features for specifying state machines of active objects. In addition to the decomposition of composite states into sequential substates, concurrent substates may be defined, which are usually associated with concurrent execution. It is often the case, that the substates execute their own threads, but this is not necessarily so. The regions for the concurrent substates are separated by dashed lines. The same notation may be used for decomposing composite states into mutual exclusive substates, where only exactly one of the substates can be active at a given instant (i.e. sequential execution). Which interpretation has to be chosen is specified by the attribute isConcurrent of class CompositeState in the UML metamodel. In the model of the system the value of isConcurrent can be shown as a property string (e.g. {isConcurrent = true}). Furthermore, it is possible to specify complex transitions which indicate the forking or joining of control flows. We will use complex transitions in our example activity diagrams, which are a special case of statecharts (Eriksson and Penker (1998), Rational Software Corp (1997b)).

3.1 Modeling a Simple Example of Multithreading

Our first example describes a Java program consisting of three parallel threads of execution. The main thread first leads to the creation of an object test and the invocation of method beginTest() of that object. Within this method, two new thread objects of class ThrEx1 are created and the threads are started. From that point on the execution of these threads occurs in parallel. Each thread performs its run method, where a loop is executed, whose upper bound has been passed—together with the thread's name—as an argument to the constructor of class ThrEx1. Within the body of the loop the thread simply prints out its name and the value of the iteration counter i, and calls method work(). The busy waiting implemented in this method is used instead of an invocation of sleep, because this guarantees, that the thread will not change its state. The Java Virtual Machine (JVM) finally exits, when all three threads (i.e. thr1, thr2 and main) have terminated.

This simple example may serve to present some more basic modeling elements of the UML in general and for expressing concurrency in particular. Figure 2 shows a class diagram representing the static design of the program as it could have been developed before the implementation. Figure 3 is a dynamic view of the program modeled as a sequence diagram, and Figure 4 expresses similar information within a collaboration diagram. These diagrams are discussed in the following subsections. Listing 1 on the following page shows the Java program code for the simple multithreading example.

Listing 1:

```
// ThrEx1.java

import java.io.*;

class ThrEx1 extends Thread {
    static PrintWriter out = new PrintWriter(System.out, true);
    int num;
    ThrEx1(String name, int num) {
        super(name);
        this.num = num;
    }
    public void run() {
        for (int i = 0; i < num; ) {
            out.println(getName() + ": " + ++i);
            work();
        }
    }
    void work() {
        long t = System.currentTimeMillis() + 500;
        while (System.currentTimeMillis() < t)
            ;
    }
}
```

```
// ThrTest1.java

class ThrTest1 {
    private ThrEx1 thr1, thr2;
    public void beginTest() {
        thr1 = new ThrEx1("Thread 1", 4);
        thr2 = new ThrEx1("Thread 2", 6);
        ThrEx1.out.println("Ready to start two Threads");
        thr1.start();
        thr2.start();
        ThrEx1.out.println("Started two Threads");
    }
    public static void main(String[] args) {
        ThrTest1 test = new ThrTest1();
        test.beginTest();
    }
}
```

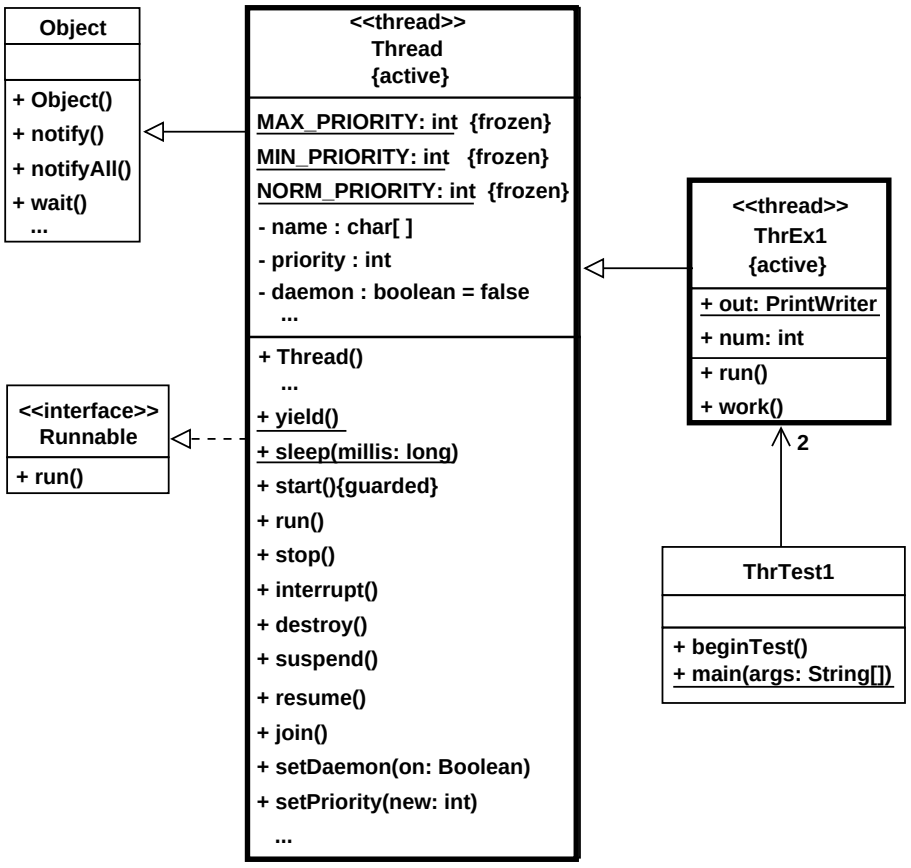

Figure 2: Simplified class diagram for Listing 1

3.1.1 Visualizing Static Aspects in Class Diagrams

Like many other OO-methods, the UML uses class diagrams to show the static structure of the system under construction, containing classes and types, their relationships to each other, and their internal structure. Although dynamic diagrams are more important regarding the modeling of concurrency, some basic information may be placed in class diagrams also.

Figure 2 shows the structure of the program in Listing 1. In the center of attention is class Thread, which is not defined in the listing, but originates from the Java class library. It extends class Object (the generalization relationship is shown as a line with a hollow triangle), which is the root of all Java classes, and implements the Java interface Runnable (the implementation relationship is shown as a dashed line with a hollow triangle). We have restricted ourselves to show only the most relevant attributes and operations of Thread and Object in the diagram. Thread is extended by class ThrEx1, which is defined in Listing 1. Likewise, class ThrTest1 is shown, which contains the entry point to the program, the static function main. Note that class-scope attributes and operations are underlined in UML diagrams. The arrow from ThrTest1 to ThrEx1 and the multiplicity 2 indicate, that each object of ThrTest1 knows exactly two objects of class ThrEx1, whereas these objects do not know their counterpart.

In UML, active classes and objects are shown as symbols with heavy borders. They are often shown as composites with embedded parts, which is not the case in Figure 2. The heavy border

is a notational convenience which may be seen as the graphical representation of a stereotype for classes, indicating that they are active. Stereotypes are a built-in extensibility mechanism of UML, because they introduce new classes of modeling elements to the set provided by UML at modeling time. Stereotypes have to be applied to existing modeling elements in the metamodel of UML and represent subclasses of them with probably additional constraints. The standard notation for a stereotype is a string in guillemets. In Figure 2, the stereotype ≪thread≫, which is predefined in UML, indicates, that the active class Thread is implemented as a thread, and not as a process. Analogously, ≪interface≫ classifies Runnable as an interface, and not a class. Optionally, a stereotype may be associated with a graphic icon or marker, and the representation of the base modeling element may even be replaced completely by the associated stereotype symbol. Another way of indicating that a class or an object is active, is to use the property keyword {active}, which specifies, that the class attribute isActive of metamodel class Class holds true. We used all these mechanisms in the class diagram, although at least the use of heavy borders and the use of the property string {active} are redundant.

3.1.2 Visualizing Dynamic Aspects in Sequence Diagrams

Sequence diagrams enable us to show the interaction of objects, i.e. the messages they exchange, arranged in time sequence. Figure 3 shows the objects created by the program in Listing 1 and the message flow between them, which has to be read from top to bottom. The vertical dashed lines below the object rectangles represent the lifelines of the objects, which indicate the existence of the object at a particular time. Message arrows creating an object end right next to the object rectangle. The thin, vertical rectangles show activations of the respective object. Activations represent the period of time during which an object performs an action or, in the case of procedural code, a subordinate method is active, which may belong to some other object. Concurrent objects often have execution control during their whole lifetime, performing their run activity while both sending and receiving messages. In the case of concurrency, operations of other objects are not relevant to the activation of an object and the entire lifeline may be shown as an activation, if the distinction between direct computation and indirect computation (by a nested procedure) is unimportant.

The physical location of the arrows in the sequence diagram shows the relative sequences of messages. In the presence of concurrent threads of control, sequence numbers may be used to identify those threads. We do not use sequence numbers in Figure 3, but use them in the corresponding collaboration diagram (see Figure 4).

As mentioned before, the UML supports different kinds of communication between objects in a system. For example, messages can be implemented as procedure calls, as the sending of signals between active objects, as the raising of a specific event, and so on. In UML, different kinds of communication are distinguished by the use of different types of arrows for messages. The filled solid arrowhead predominating in Figure 3 stands for a method call or nested flow of control. In the case of concurrently active objects, it indicates wait semantics, i.e. an object sends a signal and waits for a nested sequence of behavior to complete. Half stick arrowheads, like the ones used in our example for the start() messages, stand for asynchronous flows of control where the sender does not wait for the receiver to deal with the message. Stick arrowheads should be used to express a flat flow of control without describing any details about the communication. Returns may be shown by dashed arrows with stick arrowheads. They may be omitted when

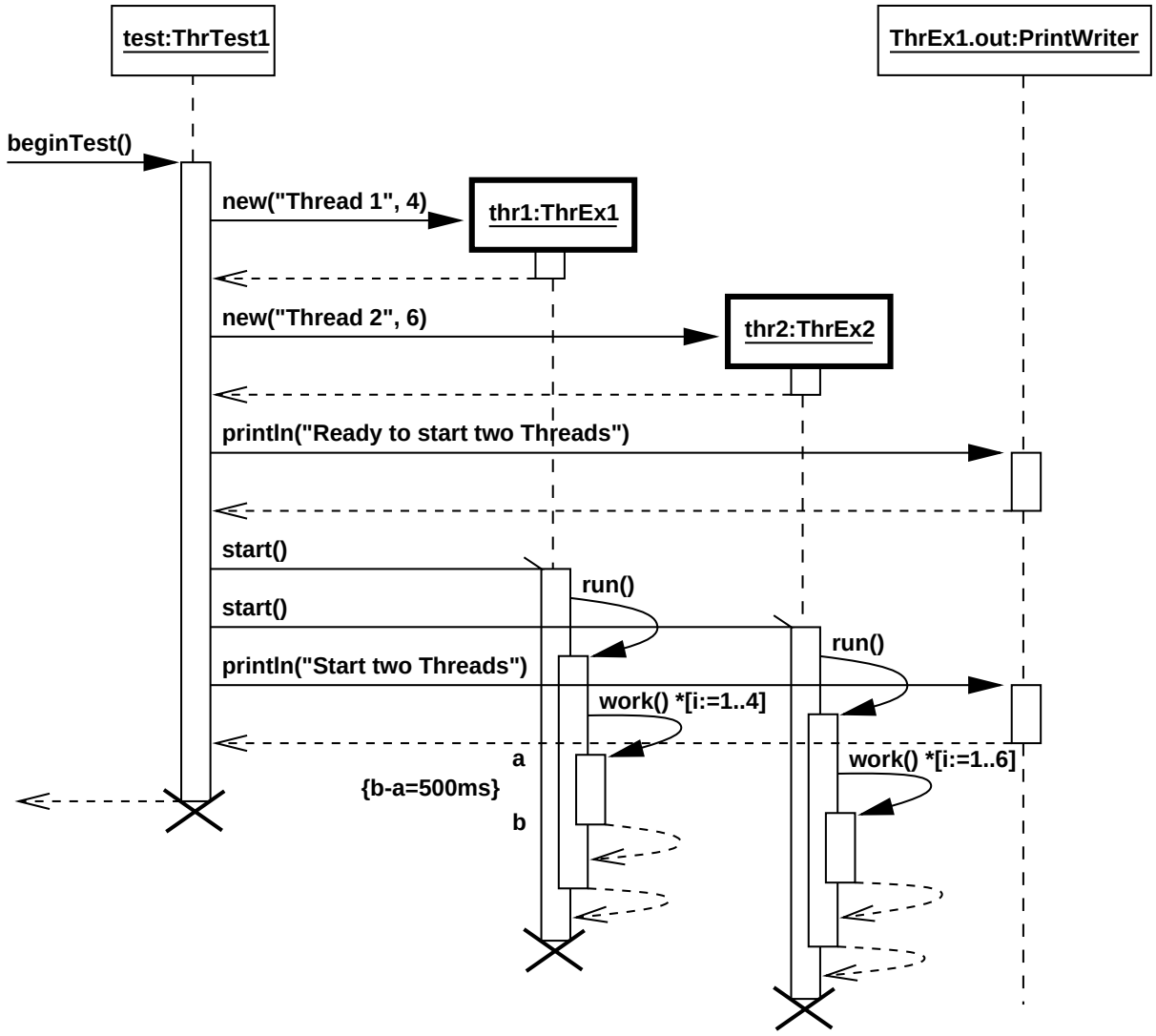

Figure 3: Sequence diagram for Listing 1

purely procedural flow of control occurs, but for parallel processing and asynchronous messages they should be shown explicitly.

In Figure 3, the procedural call of method beginTest() is shown, which leads to an activation of object test. The sequential creation of two thread objects and the printing of a string follow; return arrows indicate the completion of the respective operations. The sending of start() messages to the thread objects is asynchronous, and test continues its own execution by printing the Started two threads info and destroying itself after the completion of the printing (indicated by a bold cross). At the same time, the threads execute their run() method, which calls work() several times. The staggered activations indicate the resulting stack frame. *[i:=1..n] is an iteration clause which simplifies the presentation of the sequential execution of method calls in an iteration. In addition, UML offers the opportunity to specify that the methods invoked are executed concurrently. In that case, the star would have to be followed by a double vertical line for parallelism: *||.

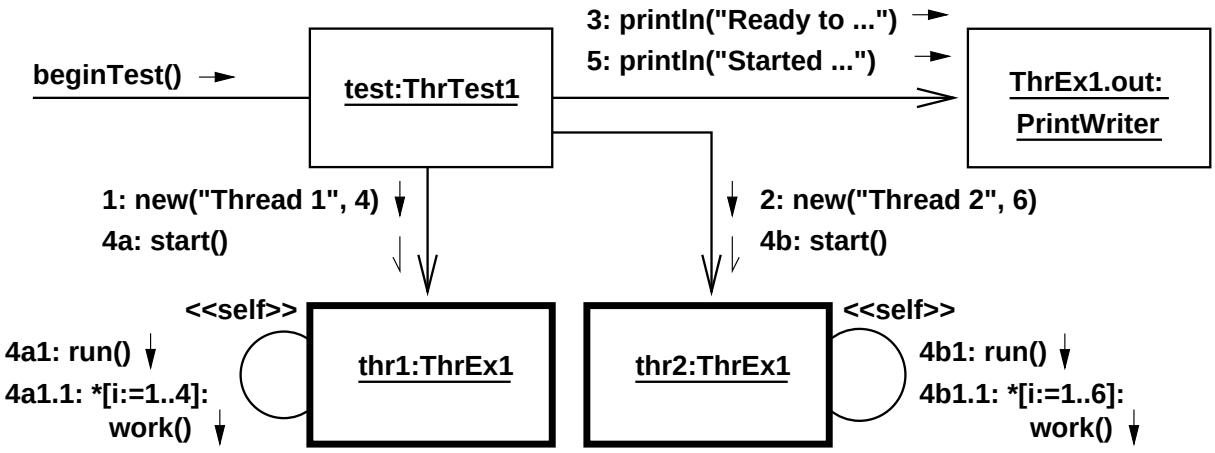

Figure 4: Collaboration diagram for Listing 1

Another feature of UML sequence diagrams, which is important for modeling real-time systems, is the possibility to label the diagram with timing marks and timing constraints. In our example, the constraint {b-a=500ms} prescribes the duration of the execution of method work(). A feature not shown in the example is branching. Branches are labeled by guard conditions and are shown as multiple arrows leaving a single point. This may be used to visualize concurrency, but only if the guard conditions are not mutually exclusive. In the latter case, conditionality is expressed in the sequence diagram, so that it takes a general form instead of an instance form.

A very important task in modeling is to ensure that the diagrams expressing different system views are consistent and integrated. There are a number of consistency rules to be followed. For example, the objects shown in the sequence diagram of Figure 3 need to have corresponding classes in the class diagram of Figure 2, and, if the classes are active, the objects in the sequence diagram need to have heavy borders just like their corresponding classes in the class diagram. Furthermore, messages arriving at an object lifeline in a sequence diagram must correspond to operations declared for that object's class in the class diagram. The adherence to these kinds of consistency rules may best be supervised by a case tool used for modeling and system development.

3.1.3 Visualizing Dynamic Aspects in Collaboration Diagrams

Collaboration diagrams are another kind of dynamic diagram in the UML. Although they express information similar to that expressed by sequence diagrams, they lay emphasis on the inclusion of structural aspects of an interaction, whereas the time sequence is no dimension of its own in collaboration diagrams. This is the reason why the use of sequence numbers in collaboration diagrams is indispensable for determining the message sequences and multiple threads of control.

Figure 4 shows a collaboration diagram for Listing 1, showing the collaboration of the four objects involved in order to execute beginTest(). The message labels placed near the object links are specified according to the following template: predecessor guard-condition sequence-expression

return-value := message-name argument list. Predecessors and guard conditions are not shown in the example, but they can be important for modeling concurrency. If predecessors are specified, the message flow cannot continue until all of the preceding message flows have occured and the guard condition is satisfied. The specification of a list of predecessors therefore represents a means of thread synchronization. The sequence-expression consists of terms of the form [integer | name][recurrence], separated by dots and ended by a colon. The terms represent levels of procedural nesting, which do not occur, if the control flow is solely concurrent. In that case, all the sequence numbers are at the same level. A message sequence is expressed by ascending integer numbers at the same level of nesting depth. In Figure 4, message println(...) with sequence number 3 follows message new("Thread 2", 6) with sequence number 2. Names, on the other hand, represent concurrent threads of control. In our example, 4a and 4b indicate the starting of the parallel execution of two threads. With the help of a recurrence string, conditional or iterative execution can be specified (see Figure 3). The types of arrows used to distinguish different kinds of communication are the same as those used in sequence diagrams. Additionally, model consistency considerations similar to those mentioned in section 3.1.2 arise.

3.2 Characteristics of Daemon-Threads

Java distinguishes between user threads and daemon threads. The difference is that the JVM does not wait for daemon threads to terminate before exiting, whereas it cannot exit as long as any user threads are still running. A thread should be set as a daemon or user thread when it is created. The kind of thread an object represents is stored in its private instance variable daemon of type boolean, which is false by default (see Figure 2). Via the public method setDaemon(boolean on) the daemon-feature of a thread object can be set and isDaemon() returns the state of the feature.

The following listing is identical with Listing 1 except for the statement making thr2 a daemon thread object (thr2.setDaemon(true);). The effect is, that the program might stop before thr2 has counted up to 6, because thr1 and the main thread have already terminated, causing the JVM to stop.

In UML design models, daemon threads should be marked as daemon threads explicitly by using one of UML's built-in extension mechanisms. For example, in object or collaboration diagrams daemon objects could be emphasized by a property string using a tagged value (e.g. {kind=daemon}). The tagged value mechanism can be used to define new model element properties in order to permit arbitrary information to be attached to models (Rational Software Corp. (1997a)).

In the sequence diagram presented in Figure 3 the modification in Listing 2 would show in two places: first, the setDaemon-message would have to be added, and second, the lifeline of thr2 would have to stop in accordance with the termination of the last user thread (here probably thr1).

Listing 2:

```
// ThrTest2.java

class ThrTest2 {
   private ThrEx1 thr1, thr2;
```

```
    public void beginTest() {
        thr1 = new ThrEx1("Thread 1", 4);
        thr2 = new ThrEx1("Thread 2", 6);
        ThrEx1.out.println("Ready to start two Threads");
        thr2.setDaemon(true);
        thr1.start();
        thr2.start();
        ThrEx1.out.println("Started two Threads");
    }
    public static void main(String[] args) {
        ThrTest2 test = new ThrTest2();
        test.beginTest();
    }
}
```

3.3 Thread Priorities

In a concurrent Java program, a scheduler decides which thread should run next, if several threads are in state Runnable. Usually, some time-slicing algorithm is used to allocate a CPU to threads in order to simulate parallel execution. Programmers may take influence on scheduling by setting thread priorities using the method setPriority(), since the scheduler prefers threads with high priorities. The range of permissible priority numbers for threads lies between Thread.MIN_PRIORITY and Thread.MAX_PRIORITY, and by default the priority is set to Thread.NORM_PRIORITY. Listing 3 gives an example of setting priorities for threads.

There is no built-in modeling element for specifying priorities of active objects in UML, but the extensibility mechanisms of UML can come to assistance again. The modeler should simply define a new property for active classes in the form of a tagged value priority which can be assigned the appropriate priority values in UML diagrams (Eriksson and Penker (1998)). Since it is normally impossible to rely on priorities in regard to the actual scheduling dynamics, the impact of priorities on the running times of the threads involved cannot and should not be shown in dynamic diagrams.

Listing 3:

```
// ThrTest3.java

class ThrTest3 {
    private ThrEx1 thr1, thr2;
    public void beginTest() {
        thr1 = new ThrEx1("Thread 1", 4);
        thr2 = new ThrEx1("Thread 2", 6);
        thr1.setPriority(Thread.MIN_PRIORITY);
        thr2.setPriority(Thread.MAX_PRIORITY);
        thr1.start();
        thr2.start();
    }
```

```
    public static void main(String[] args) {
        ThrTest3 test = new ThrTest3();
        test.beginTest();
    }
}
```

3.4 Modeling Synchronization with Activity Diagrams

One of the main tasks in designing concurrent systems is to coordinate multiple threads with each other in such a way that the system remains consistent and the intended semantics are preserved. For example, different active objects have to be prevented from trying to modify or access a shared resource simultaneously. UML's mechanisms for expressing synchronization semantics are presented in this and the following subsection.

The first example of synchronization (Listing 4) shows the simplest form of coordination between concurrent threads: A thread creates other threads and waits for their termination before it continues its own execution. In Java, this can be achieved by having the creator thread send join() messages to its child threads.

Listing 4:

```
// ThrTest4.java

class ThrTest4 {
    private ThrEx1 thr1, thr2;
    public void beginTest() throws InterruptedException {
        thr1 = new ThrEx1("Thread 1", 4);
        thr2 = new ThrEx1("Thread 2", 6);
        ThrEx1.out.println("Ready to start two Threads");
        thr1.start();
        thr2.start();
        thr1.join();
        thr2.join();
        ThrEx1.out.println("Started two Threads");
    }
    public static void main(String[] args) {
        ThrTest4 test = new ThrTest4();
        test.beginTest();
    }
}
```

The effect of joining can be illustrated very well with the help of activity diagrams (Fowler (1997)). Activity diagrams are a special kind of state diagram, containing so-called action states and transitions which are (usually) triggered by completion of the actions in the source action states. Activity diagrams have to be associated with a class, an operation or a use case (Rational Software Corp. (1997a)). In Figure 5, the implementation of method beginTest() of

Figure 5: Activity diagram illustrating forks and splits in Listing 4

class ThrTest4 is shown with actions expressed in Java language code (alternatives are natural language or pseudo code). The three areas, divided by dashed lines, are called swimlanes, which are a kind of package for organizing responsibility for activities. They are used here to show the responsibilities of the different threads created during program execution.

To show the splitting and joining of control flow, we need complex transitions. A complex transition, drawn as a synchronization bar, either splits control into simultaneously executed

threads or unifies concurrent threads into a single one. There is a correspondence between parallel parts of control flow in activity diagrams and concurrent substates in normal state diagrams. "The dynamic semantics of activity models can be expressed in terms of state machines. This means that the process structure of activities formally must be equivalent to orthogonal regions (in composite states)." (Rational Software Corp. (1997b))

The diagram illustrates how the thr1.start() message leads to the splitting of control flow and the parallel execution of thr1.run() and thr2.start(). The same happens again with the starting of the second thread. After performing thr1.join(), the main thread becomes Not Runnable until thr1.run() is complete, i.e. the thread named Thread 1 has terminated (which leads to a state transition back to Runnable, see Figure 1). Analogously, the main thread waits for the termination of the second thread. After it has become Runnable again and a CPU has been assigned to it (i.e. it has become active), the final output is made.

3.5 Object-Level and Class-Level Thread Synchronization

Java integrates a means of synchronizing multiple threads at the language level by providing the synchronized modifier. This modifier may be used to mark some portion of code (usually class or instance methods) which needs a lock in order to run. Locks (or monitors) are owned by classes or objects in order to keep sections of code from running at the same time as others. A thread can take temporary ownership of a lock and release it later so that another thread can become the owner of the same lock. By owning a lock, a thread hinders all other threads from invoking other synchronized methods defined for an object or class that the lock belongs to. Listing 5 shows an example of using a lock at the object level.

The UML metamodel predefines a property concurrency for operations, which may take the values sequential, guarded, or concurrent in order to specify the concurrency semantics of a class-scope or instance-scope operation. The property has to be shown as a tagged value in UML diagrams (see Figure 6 as an example). The value sequential specifies that the integrity of the class/object cannot be guaranteed if simultaneous calls occur, i.e. active clients calling the same sequential operation of a passive class/object have to ensure by themselves, that only one call is outstanding at any time. With the value guarded, the semantics of Java's synchronized modifier can be expressed: Guarded operations allow multiple calls from concurrent threads simultaneously, but block all threads as long as another thread executes the operation. In Java, this means that an active thread trying to invoke a synchronized method which is already occupied by another thread stays Runnable, but becomes inactive. In this situation, the programmer has to take care of avoiding deadlocks due to simultaneous blocks. Specifying an operation as concurrent indicates, that multiple threads may safely work in parallel on that object/class without corrupting its state (Rational Software Corp. (1997b)).

Listing 5:

```
// Worker1.java

class Worker1 {
   synchronized void work() {
      long t = System.currentTimeMillis() + 500;
```

```
        while (System.currentTimeMillis() < t)
            ;
    }
}
```

```
// ThrEx2.java

import java.io.*;

class ThrEx2 extends Thread {
    static PrintWriter out = new PrintWriter(System.out, true);
    int num;
    Worker1 worker;
    ThrEx2(String name, int num, Worker1 worker) {
        super(name);
        this.num = num;
        this.worker = worker;
    }
    public void run() {
        for (int i = 0; i < num; ) {
            out.println(getName() + ": " + ++i);
            worker.work();
        }
    }
}
```

```
// ThrTest5.java

class ThrTest5 {
    public static void main(String[] args) {
        Worker1 w = new Worker1();
        ThrEx2 thr1 = new ThrEx2("Thread 1", 4, w),
            thr2 = new ThrEx2("Thread 2", 6, w);
        thr1.start();
        thr2.start();
    }
}
```

3.6 A More Complex Example: Producer, Mediator and Consumer

Our last example will combine a considerable number of the concepts described before in this paper. We discuss the implementation of a simple Producer/Consumer-Pattern, where a producer supplies a consumer with an integer value via a shared mediator object. To allow efficient execution, both producer and consumer block (i.e. become Not Runnable) if the resource is not

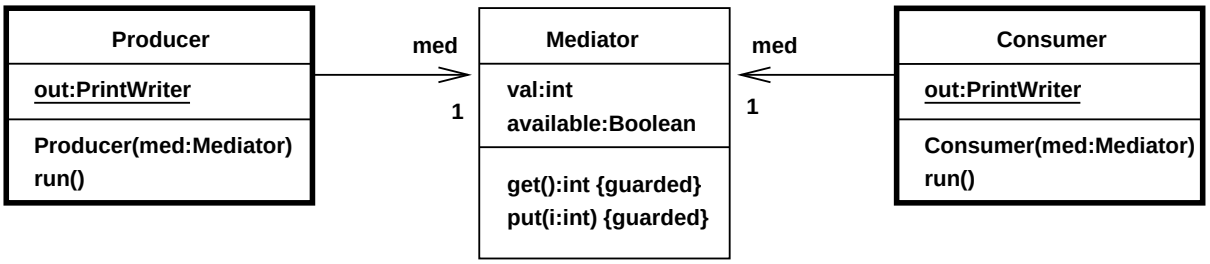

Figure 6: Class Diagram for Producer/Consumer-Listing

available and wake up (i.e. become Runnable again) when being notified by their counterpart. Furthermore, the mediator object uses synchronized methods to avoid interference of the two threads when its methods are called.

Figure 6 shows the class diagram corresponding to Listing 6, containing the active classes Producer and Consumer and the passive class Mediator. Mediator's methods are guarded, which indicates that they will be implemented as synchronized in Java code. The synchronization mechanisms in this example are quite complex, so we have visualized them with the help of an activity diagram (see Figure 7). Amongst pseudo states, action states, transitions and complex transitions we have made extensive use of another UML feature for activity diagrams (and state diagrams in general): decisions. Decisions are used to indicate different possible transitions to successor states in dependency of a boolean guard condition. The presence of decisions can be made explicit by the use of diamond shaped icons.

Listing 6:

```
// Producer.java

import java.io.*;

class Producer extends Thread {
    static PrintWriter out = new PrintWriter(System.out, true);
    Mediator med;
    Producer(Mediator med) {
        this.med = med;
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            med.put(i);
            out.println("Producer put: " +i);
            try {
                sleep((int)(100.0*Math.random()));
            } catch (InterruptedException ignore) { }
        }
    }
}
```

```
// Consumer.java

import java.io.*;

class Consumer extends Thread {
    static PrintWriter out = new PrintWriter(System.out, true);
    Mediator med;
    Consumer(Mediator med) {
        this.med = med;
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            out.println("Consumer got: " + med.get());
            try {
                sleep((int)(1000.0*Math.random()));
            } catch (InterruptedException ignore) { }
        }
    }
}
```

```
// Mediator.java

class Mediator {
    int val;
    boolean available;
    synchronized int get() {
        while (!available)
          try {
              wait();
          } catch (InterruptedException ignore) { }
        notify();
        available = false;
        return val;
    }
    synchronized void put(int i) {
        while (available)
          try {
              wait();
          } catch (InterruptedException ignore) { }
        notify();
        val = i;
        available = true;
    }
}
```

```
// ThrTest6.java

class ThrTest6 {
    public static void main(String[] args) {
        Mediator med = new Mediator();
        Producer prod = new Producer(med);
        Consumer con = new Consumer(med);
        con.start();
        prod.start();
    }
}
```

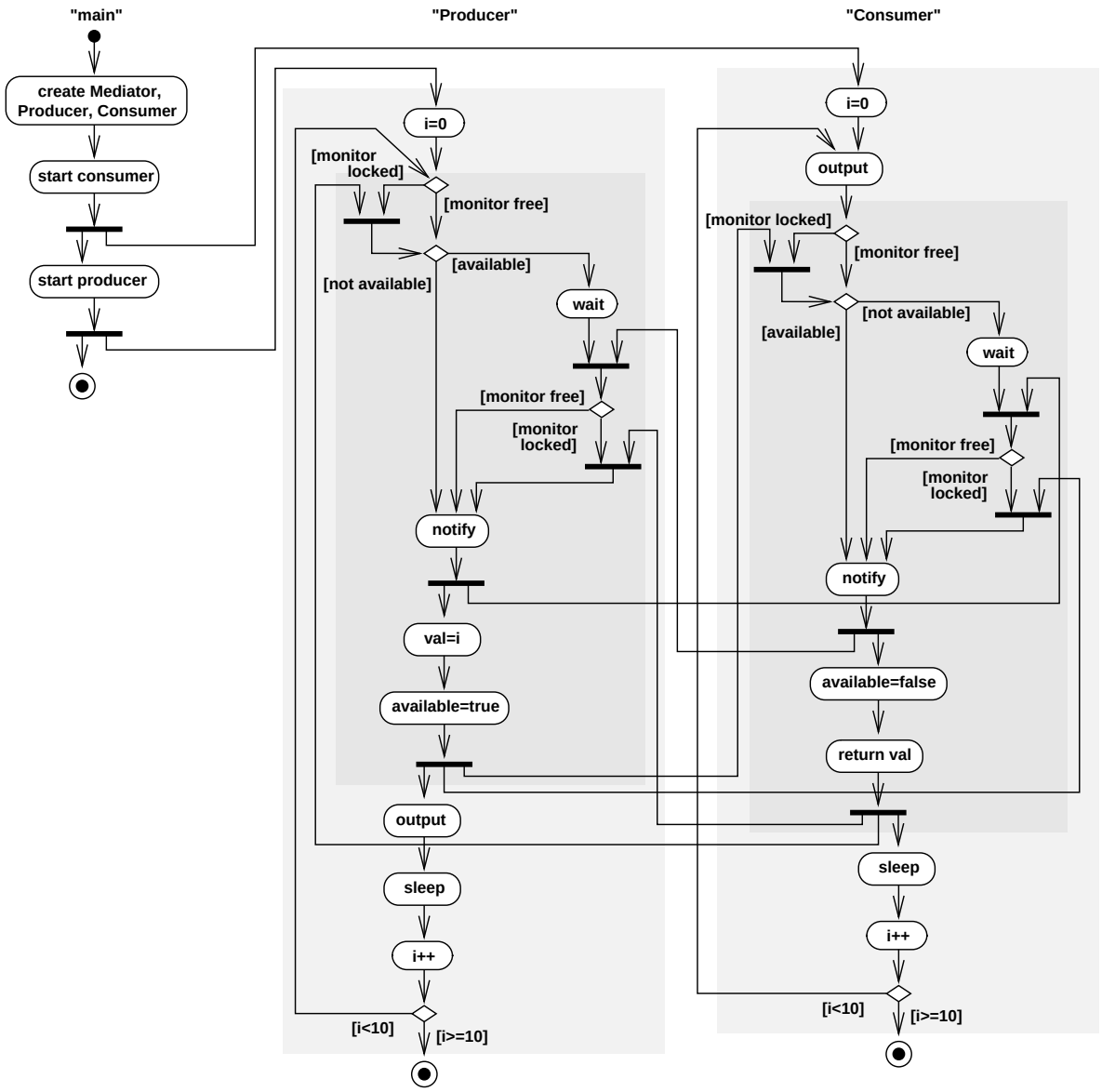

Figure 7: Activity diagram of consumer/producer problem with pseudo code

The example activity diagram uses pseudo code to describe the actions in the control flow of the Producer/Consumer-Pattern. Again, the different threads can be identified by different horizontal regions (the dashed lines are left out here). In order to be able to show the complex intertwining of the threads, we decided to reveal the internal flows of methods run(), put(int i), and get(). The light grey regions depict the run() method's control flow, and the dark grey regions show the control flows of put(int i) and get(), respectively. put(int i) and get() are responsibilities of the mediator object, which are not separated into a distinct swimlane here in order to show the intertwining more directly. When analyzing the code of Listing 6 and in particular the implementation of method put(int i), the reader may at first be surprised at the fact, that notify() is called before a value is assigned to val, because notify() leads to changing the state of the consumer thread to Runnable, so that it tries to read the new value. As can be seen from the activity diagram, the consumer thread may of course not execute method get() until the producer thread has released the lock of object med, and this is not done until the producer thread completes put(), leaving behind a correct state of med for the consumer thread to continue.

4 Conclusion

This paper has provided the reader with a basic understanding of Java threads and their modeling with the help of the UML. It has been shown that, in contrast to many other current modeling languages, UML includes the basic mechanisms for expressing concurrency, which are sufficient for standard applications with a small number of threads of control. Nevertheless, it has to be stated clearly that the UML is a general-purpose modeling language which, of course, has its limitations. Therefore, the UML camp agreed to build specialized, domain-specific variants of UML (UML Variants and UML Extensions: see Rational Software Corp. (1997c)) to serve special purpose domains. One important variant of UML under development is UML-RT (ObjecTime (1997)), which is based on ObjecTime's ROOM methodology and adds value for modeling real-time domain-specific applications requiring concurrency, distribution, high reliability, availability, serviceability, and so forth. In order to analyze and design industrial strength real-time systems with a high degree of parallelism and other real-time requirements, such as timing constraints, reliability, performance etc., one has to consider using UML-RT which might fit the purpose better than the standard version of UML. On the other hand, this kind of UML variant will further increase the level of complexity inherent in the modeling language—a problem, even the UML core suffers from—and, at the same time, will decrease its ease of use and its understandability for non-experts.

References

BOOCH, G. (1994): Object-Oriented Analysis and Design with Applications. 2nd edn. Benjamin/Cummings, Redwood City, California.

BOONE, B. (1996): Java Essentials for C and C++ Programmers. Addison-Wesley, Reading, Massachusetts.

BOONE, B. (1997): Java Certification for Programmers and Developers. McGraw-Hill, New York.

COAD, P. and MAYFIELD, M. (1997): Java Design – Building Better Apps & Applets. Yourdon Press Computing Series, Prentice Hall, New Jersey.

COAD, P. and YOURDON, E. (1991): Object-Oriented Analysis. 2nd edn. Prentice Hall, Englewood Cliffs, New Jersey.

ERIKSSON, H.-E. and PENKER, M. (1998): UML-Toolkit. Wiley Computer Publishing, New York.

FOWLER, M. and SCOTT, K. (1997): UML Distilled – Applying the Standard Object Modeling Language. Addison-Wesley, Reading, Massachusetts.

GOMAA, H. (1993): Software Design Methods for Concurrent and Real-Time Systems. SEI Series in Software Engineering. Addison-Wesley, Reading, Massachusetts.

LEA, D. (1997): Concurrent Object Models – Design Principles and Patterns. Addison-Wesley, Reading, Massachusetts.

OBJECTIME (1997): http://www.objectime.com/new/uml-rt/index.html

RATIONAL SOFTWARE CORP. (1997a): UML Notation Guide. Version 1.1, 1 September 1997, Rational Software Corp. et al.

RATIONAL SOFTWARE CORP. (1997b): UML Semantics. Version 1.1, 1 September 1997, Rational Software Corp. et al.

RATIONAL SOFTWARE CORP. (1997c): UML Summary. Version 1.1, 1 September 1997, Rational Software Corp. et al.

RITCHEY, T. (1995): Programming with Java! New Riders Publishing, Indianapolis, Indiana.

SCHADER, M. and RUNDSHAGEN, M. (1996): Objektorientierte Systemanalyse – Eine Einführung. Springer-Verlag, Heidelberg.

STEIN, W. (1995): Objektorientierte Analyse für nebenläufige Systeme: Die Methode MORE/RT. Reihe Forschung in der Softwaretechnik, Band 2, Herausgeber: H. Balzert. BI Wissenschaftsverlag, Mannheim.

SUN MICROSYSTEMS, INC. (1997): http://www.javasoft.com/docs/