

# Design and Evaluation of a Diagrammatic Notation to Aid in the Understanding of Concurrency Concepts

Shaohua Xie<sup>1</sup>, Eileen Kraemer<sup>1</sup>, R.E.K. Stirewalt<sup>2</sup>  
*The University of Georgia<sup>1</sup>, Athens, GA*  
*Michigan State University<sup>2</sup>, East Lansing, MI*  
{shaohua, eileen}@cs.uga.edu, stire@cse.msu.edu

## Abstract

*It is generally accepted that concurrency can be difficult for students to reason about and to manage. While some studies provide insight into the nature of these difficulties[6], work remains to be done in understanding the aspects of learning about concurrency that are most difficult, and in developing approaches to dealing with this problem. We have conducted instructor interviews and an observational study of students, identified several key difficulties that students encounter, and developed a diagram that we believe will be an aid to understanding and problem-solving. We present the diagram and results of an initial user evaluation.*

## 1. Introduction

Concurrency and synchronization concepts are generally seen as difficult for novices to grasp [5, 2, 9, 3, 7]. In teaching students about concurrency, instructors usually present motivating examples and show students the code for low-level, unstructured synchronization primitives such as *wait* and *notify* operations on condition variables. Instructor's goals include: 1) That students will understand how the invocation of a synchronization primitive affects the state of operating system data structures such as condition queues, ready queues, lock ownership, etc; and 2) That students will be able to reason correctly about thread interactions and synchronization behavior.

Through the code and examples, students typically are able to achieve the first objective: to see the potential effects of a thread on the OS data structures and states. However, many students find it difficult to achieve the second objective: to apply this knowledge when reasoning about the behavior of threads that synchronize using these primitives. This ability to reason about the synchronization behavior of threads requires the student to understand not only the individual actions of a thread or the implementation of a particular primitive, but to also comprehend the interactions between threads, their interleavings, and

the effects of the execution of these synchronization primitives by one thread on the operating system data structures and states and thus on other threads. Applying the terminology of Bloom's taxonomy [1], the first objective may be described as comprehension (level 2) behavior, while the second objective requires applying this knowledge and is regarded as an application (level 3) behavior. This higher-level objective is considered more difficult.

Kolikant [6] has performed empirical studies of students learning about concurrency. His results show that novice students develop pattern-based techniques to solve synchronization problems and that these students also have trouble in solving non-familiar synchronization problems, perhaps as a result of their reliance on those pattern-based approaches. It seems that students cannot reason about thread interaction in novel situations without a deep understanding of how the effects of the low-level primitives on OS states and data structures indirectly affect thread synchronization.

External representations of concurrency concepts should aid students in achieving both objectives. That such external representations can interact with internal representations to improve problem-solving is well-supported by work in distributed cognition [10]. Pancake states that purely textual representations may be inadequate to express complex concurrent programming situations [8]. Thus, visual representations are needed.

In this paper, we (1) describe an instructor survey and observational study through which we identified a core set of difficulties that students encounter in learning about concurrency; (2) list those difficulties; (3) present a refinement of the UML sequence diagram and show how its use can help students; (4) describe the results of a subjective survey; (5) discuss our plans for refinement of the diagram and a more in-depth, objective study of the utility of these diagrams.

## 2. Interviews and observational study: What do students find difficult?

We conducted instructor interviews and an observational study to identify the difficulties that students experience when learning about concurrency. Three instructors of courses that deal with concurrency concepts were interviewed and asked to identify the nature of the difficulties students encountered.

To confirm the common problems pointed out by the instructors and to search for additional student difficulties, we conducted an observational study. During five weeks of class sessions, we observed student behavior and took notes about each instructor's presentation, the students' questions and the student responses to the instructors' questions. Common problems we identified include:

1. Thread interleavings are difficult for students to comprehend.
2. Students often forget that context switches can happen when the thread is in a monitor or critical section and have trouble correctly applying that knowledge when they do remember.
3. It is common for instructors to use ad-hoc sketches to describe concurrent program executions. However, students often find it difficult to record the details and explanations for later review of the reasoning behind the sketches.
4. Students have trouble reasoning about *why* the implementations of synchronization primitives lead to correct synchronization behavior.
5. Students often find it difficult to choose appropriate synchronization mechanisms and primitives to meet certain synchronization goals.

## 3. Our refinement of the sequence diagram

We have developed a refinement of the UML sequence diagram that attempts to address the first three difficulties, and may help students to develop a sufficiently deep understanding of synchronization behavior to assist in the last two difficulties. In the following sections, we describe our diagram, applied to a monitor solution to the readers-writers problem [4].

### 3.1. Readers-writers: monitor solution

The readers-writers problem is a classic synchronization problem in which two distinct classes of threads exist, **reader** and **writer**. Multiple **reader** threads can enter the *critical section* simultaneously. However, no other **writer** thread, nor any **reader** thread, may be present in the critical section while a given **writer** thread is present. We present Java-like pseudocode of a monitor solution to the readers-writers problem in Figure 1.

Figure 2 presents the implementations of the *wait*, *notify* and *notifyAll* method calls on condition variables. Note: (1) each condition variable maintains a “wait set” onto which threads may be placed to await resumption when some other thread invokes *notify* or *notifyAll* for this condition variable; (2) *wait*, *notify* and *notifyAll* are methods of class Object; thus the “release lock” statement releases the lock on the monitor object and does not affect the condition variable passed in as a parameter (3) *Thread.currentThread()* returns the currently executing thread and *suspend()* causes the target thread to suspend execution until it is explicitly resumed by another thread.

```
class Database extends Object {
    private int numReaders = 0;
    private int numWriters = 0;
    private ConditionVariable OKtoRead = new
        ConditionVariable();
    private ConditionVariable OKtoWrite = new
        ConditionVariable();
    public synchronized void startRead() {
        while (numWriters > 0)
            wait(OKtoRead);
        numReaders++;
    }
    public synchronized void endRead() {
        numReaders--;
        notify(OKtoWrite);
    }
    public synchronized void startWrite() {
        while (numReaders > 0 || numWriters > 0)
            wait(OKtoWrite);
        numWriters++;
    }
    public synchronized void endWrite() {
        numWriters--;
        notify(OKtoWrite);
        notifyAll(OKtoRead);
    }
}
```

Figure 1: A monitor solution to the readers-writers problem

```
wait(ConditionVariable cond) {
    put the calling thread on the “wait set” of cond;
    release lock;
    Thread.currentThread.suspend();
    acquire lock;
}
notify(ConditionVariable cond){
    choose t from wait set of cond;
    t.resume();
}
notifyAll(ConditionVariable cond){
    forall t in wait set of cond;
    t.resume()
}
```

Figure 2: The implementation code for “wait”, “notify” and “notifyAll”.

### 3.2. Graphical attributes of the diagram

We have extended the existing sequence diagram notation in three ways: (1) We use two object states, *locked* and *unlocked*, to indicate the state of the monitor after its lock has been acquired or released by

a thread; (2) We use colored activation bars to indicate the status of a thread: *running* (green), *ready* (yellow) or *suspended* (red) (dark gray, light gray, and medium gray when viewed in monochrome and to accommodate color-blind users); (3) We use comments to show the state of the counting variables inside the monitor (in this case the instance of class Database).

Figure 3 presents our refinement of the sequence diagram (an enlarged version can be found at <http://www.cs.uga.edu/~shaohua/ICSE2007/readers-writers.pdf>).

### 3.3. A walk-through of a program execution using the diagram

The numbers 1-11 shown to the left of the diagram are keyed to the following description:

1. Initially, a **reader** thread and a **writer** thread are created “simultaneously”. The **reader** thread is scheduled first; thus its activation bar is colored dark green (darkest gray). The **writer** thread is ready (but not running) thus its activation bar is colored yellow (lightest gray).
2. The **reader** thread invokes the *startRead()* monitor routine and is able to obtain the lock, as indicated by the appearance of the *locked* object state in the diagram.
3. A context switch occurs while the **reader** thread is executing *startRead()*. The **writer** thread now runs (**writer**’s bar changes from yellow to green as **reader**’s bar changes from green to yellow).
4. The **writer** thread invokes *startWrite()* and attempts to enter the monitor. However, the monitor lock is held by the **reader** thread and the **writer** thread suspends (bar changes from green to red). The **reader** thread resumes (bar changes from yellow to green). The **reader** thread sets *numReaders* to 1 (comment bubble), releases the lock (*unlocked* state) and returns from *startRead()*. Note that at the point at which the **reader** thread releases the lock, the **writer** thread’s state changes from *suspended* to *ready* (red to yellow).
5. When another context switch occurs, the **writer** thread is able to run (green bar) and is able to obtain the monitor lock (*locked* state).
6. Since *numReaders* is non-zero, *wait(OKtoWrite)* is invoked. As depicted in Figure 2, the **writer** thread then adds itself to the wait set of *OKtoWrite* (not depicted in diagram), releases the lock (*unlocked* state) and suspends itself (bar changes to red).
7. The **reader** thread now resumes its execution (green bar). It invokes the monitor’s *endRead()* method and acquires the lock (*locked* state).
8. The **reader** thread sets *numReaders* back to 0 (comment bubble) and invokes the monitor’s *notify* method call on the condition variable *OKtoWrite*.
9. As a result of the **reader** thread’s invocation of *notify(OKtoWrite)*, the **writer** thread, which is suspended and in the wait set of *OKtoWrite*, now resumes (bar changes from red to yellow).
10. The **reader** thread returns from the *notify* method, releases the lock (*unlocked* state) and eventually finishes its execution (bar ends).

11. The **writer** thread is then scheduled (green activation bar) obtains the monitor (*locked* state), and completes its execution.

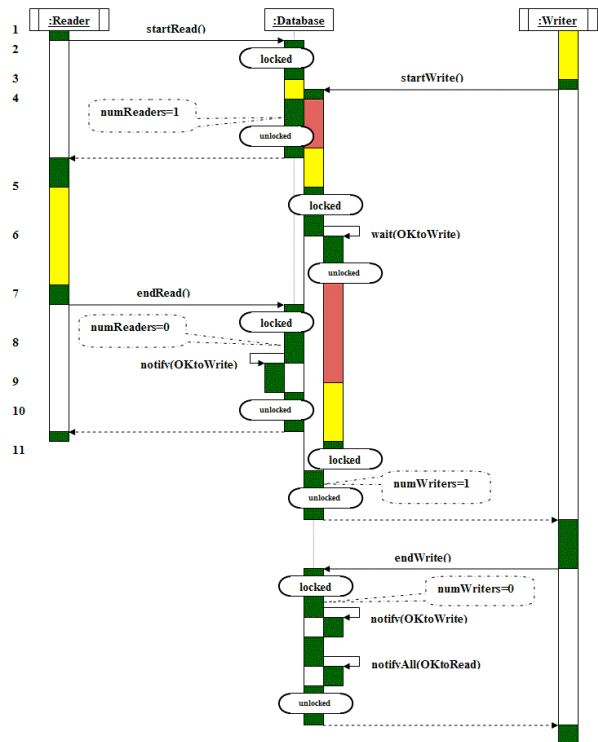


Figure 3: Our refinement of the sequence diagram for the readers-writer example.

### 3.4. Diagram assists with identified difficulties

We claim that this type of diagram can help users to overcome several of the difficulties we have identified. First, instructors may use our diagrams rather than their ad-hoc sketches during class. This will facilitate student note-taking and later review.

Thread interleavings and context switches can be easily viewed by following the trace of green activation bars. Context switches occur when the bars transition from green to other colors. Viewers can see from the diagram that context switches can occur even when the thread is in a monitor or critical section.

The diagram illustrates explicitly thread interactions and the effects of the execution of the synchronization primitives by one thread on other threads. Thus, students can more easily answer a question such as “What is the impact of the invocation of *notify* by the **reader** thread on the writer thread?” As at step 9, the status of the **writer** thread changes from *suspended* to *ready* (the bar changes from red to yellow).

Our refinement of the sequence diagram can also reveal the underlying dynamics of the synchronization primitives that are typically hidden from programmers.

For instance, at step 5, when a context switch occurs, the writer thread enters the monitor (obtains the lock). What prevents the writer from writing while the reader is still reading? The writer sees that *numReaders* is 1, and suspends on *wait(OKtoWrite)*. Why doesn't this cause a deadlock? The **writer** thread releases the monitor lock (the *unlocked* state) before it suspends.

It is also possible to use these diagrams to ask *why* a thread is suspended. Is it suspended because it can't get into the monitor or because it is waiting on some condition variable? If it is waiting on a condition variable, the suspension will have begun during an activation of *wait* and should involve a transition from green to red that occurs at the same time that an *unlocked* state is entered.

#### 4. A subjective user study

We conducted a subjective user study of our diagram. Participants were graduate students in the Computer Science Department at the University of Georgia who had recently been taught about semaphores and monitors.

We first presented the monitor solution to the readers-writers problem and the figures that appear in this paper. We "walked through" the event sequence described in Section 4. We then asked the student to fill out a short survey, which can be found at

<http://www.cs.uga.edu/~shaohua/ICSE2007/survey.pdf>

The collected results are encouraging. All of the students agreed or strongly agreed that our refinement of the sequence diagram is helpful in clarifying when threads enter and exit a monitor, which threads are actively running at any given time, illustrating the interactions between threads in a single program trace, and facilitating their understanding of the inherent mechanisms of the monitor.

#### 5. Conclusions and Future work

We have conducted formative design studies and identified some of the major difficulties that students encounter. We designed and implemented a refinement of the UML sequence diagram that we believe will be helpful in addressing these difficulties. This belief is supported by our survey feedback.

One potential criticism of these diagrams is that while lock states and thread states are easily perceived from the diagrams, the values associated with condition variables are not directly represented; their values must be inferred. We plan to develop and evaluate another version of these diagrams, in which these values are directly represented as object states.

Also of interest is to investigate how to visualize execution "in the critical section" and what types of questions such a display would be helpful in

answering. Finally, another interesting question to investigate is if it is possible to use these diagrams to verify whether the behavior depicted conforms to a safety invariant. – for example, to verify that "at no time will a writer be in the critical section concurrent with a reader". Exploring how to visualize program executions to enable viewers to answer a range of such questions is a challenging goal.

#### 6. References:

- [1] B. S. Bloom, *Taxonomy of Educational Objectives, Handbook 1: Cognitive Domain.*, David McKay, New York, 1956.
- [2] S. Carr, J. Mayo and C.-K. Shene, ThreadMentor: a pedagogical tool for multithreaded programming, *Journal on Educational Resources in Computing (JERIC)*, Vol. 3 (2003).
- [3] S.-E. Choi and E. C. Lewis, A Study of Common Pitfalls in Simple Multi-Threaded Programs, *Proceedings of the ThirtyFirst ACM SIGCSE Technical Symposium on Computer Science Education*, 2000.
- [4] P. J. Courtois, F. Heymans and D. L. Parnas, Concurrent control with "readers" and "writers", *Communications of the ACM*, Vol. 14 (1971), pp. 667 - 668.
- [5] C. Exton and M. Kolling, Concurrency, objects and visualisation, *Proceedings of the Australasian conference on Computing education*, 2000, pp. 109-115.
- [6] Y. B.-D. Kolikant, Learning concurrency: evolution of students' understanding of synchronization, *International Journal of Human-Computer Studies*, Vol. 60 (2004), pp. 243-268.
- [7] C. E. McDowell and D. P. Helmbold, Debugging concurrent programs *ACM Computing Surveys (CSUR)*, Vol. 21 (1989), pp. 593-622.
- [8] C. M. Pancake, Visualization Techniques for Parallel Debugging and Performance Tuning Tools, *Parallel Computing: Paradigms and Applications*, Intl. Thomson Computer Press, 1996.
- [9] C.-K. Shene and S. Carr, The Design of a Multithreaded Programming Course and Its Accompanying Software Tools, *The Journal of Computing in Small Colleges*, Vol. 14 (1998), pp. 12-24.
- [10] J. Zhang and D. A. Norman, Representations in Distributed Cognitive Tasks, *Cognitive Science*, Vol. 18 (1994), pp. 87-122.