

Empirical Evaluation of a UML Sequence Diagram with Adornments to Support Understanding of Thread Interactions

Shaohua Xie¹, Eileen Kraemer¹, and R.E.K. Stirewalt²
University of Georgia¹, Michigan State University²
{ shaohua, eileen }@cs.uga.edu, stire@cse.msu.edu

Abstract

Programs that use multi-threaded concurrency are known to be difficult to design. Moreover, research in computer-science education suggests that concurrency and synchronization concepts are generally difficult to master. It stands to reason that comprehension tasks may be more complex for programs that employ concurrency than for sequential programs. We believe that external representations, specifically refinements to some of the popular UML modeling notations, should aid students in mastering fundamental concurrency/synchronization concepts and should enable practitioners to better comprehend the dynamically evolving nature of these programs. In this paper, we present our synchronization adorned UML (saUML) sequence diagram notation that highlights aspects of thread interactions and describe an empirical study of whether these diagrams, as opposed to purely textual representations, help students to better understand concurrent executions and concurrency concepts, as measured by their ability to answer questions about a particular execution of a multi-threaded system. A statistically significant benefit was found from the study.

1. Introduction

Programs that use multi-threaded concurrency are known to be difficult to design. Moreover, research in computer-science education suggests that concurrency and synchronization concepts are generally difficult to master [3, 4, 6, 11, 14]. It stands to reason that comprehension tasks may be more complex for programs that employ concurrency than for sequential programs. This expectation is consistent with our experience [1, 12] and the experiences of others [6, 7, 10].

Many of the difficulties that attend to concurrent-program comprehension derive from the complexities that make concurrency and synchronization difficult for students to learn in the first place. For example, the

space of thread interleavings is enormous and difficult to meaningfully visualize in its entirety. When threads synchronize, they do so indirectly, using calls to low-level system primitives that modify the state of the operating system's data structures. That students find these primitives difficult to reason about is well documented [4, 8, 9].

We believe *external representations* [16] can both aid students in mastering concurrency and synchronization concepts and enable practitioners to better comprehend the dynamically evolving nature of concurrent programs. To this end, we have been conducting empirical studies to assess the effectiveness of various representations on human performance (and retention of knowledge) during comprehension tasks involving concurrent programs. This paper describes a study, conducted at the University of Georgia in 2006. The study shows that a modest extension to the UML 2.0 sequence diagram notation [13], when used in conjunction with source code, can significantly improve understanding tasks that involve specific executions of a concurrent program when compared with using only the source code. The improvement was measured by user performance in answering scenario-based questions about those executions.

The remainder of the paper is structured as follows. We first outline some of the fundamental problems that students face when learning about concurrency and synchronization and describe prior work in the use of external representations to improve learning (Section 2). We then describe our synchronization adorned UML (saUML) sequence diagram notation (Section 3), the experimental design, and the results of the study itself (Section 4), and the conclusions we were able to draw from this work (Section 5).

2. Background

Several studies have looked at how students learn about concurrency and the kinds of problems that attend to this learning. Kolikant's research shows that

novice students often develop pattern-based techniques to successfully solve synchronization problems to avoid dealing with the dynamics of the synchronization mechanisms. These students are then unable to deal adequately with novel situations requiring synchronization [8]. Choi and Lewis [4] performed a detailed study of student programs involving concurrency and synchronization and found that over 30% contained serious design flaws (e.g., data races, deadlocks, and inappropriate locking regimes) even though the programs produced correct output when executed on the sample inputs provided.

Kramer [9] observes that the ability to think abstractly is critical for a number of activities in computer science, including program analysis and comprehension of concurrent computations. He notes that only 30-35% of adults reach the formal *operations* stage of cognitive development at which such abstract thinking is supported. The prospects for success in these activities would seem dismal for the remaining 65-70%. However, studies of distributed cognition [16] provide some hope.

Distributed cognition involves the interaction between internal representations and external representations in the performance of problem-solving tasks. *Internal representations* are described as existing "in the mind, as propositions, productions, schemas, mental images, connectionist networks, or other forms", while *external representations* exist "in the world" both as physical symbols (e.g., written symbols or the beads on an abacus) and the rules, constraints, or relations that pertain to those symbols (e.g., the relations among the symbols in a diagram or the physical constraints on the behavior of the abacus) [16]. Norman and Zhang showed that the choice of external representation affects the speed and accuracy of users engaged in such problem-solving tasks. Thus, the choice of representation is important. The extent to which "good" choices of external representation can expand the population capable of successfully performing these tasks is an open question.

Our work intends to optimize the efficiency of program comprehension in the context of concurrency and synchronization by designing and empirically evaluating external representations that are customized to address the complexities that most programmers and analysts encounter. We began by focusing on the complexities that complicate learning, with the idea that these same complexities would also complicate comprehension tasks. Instructor interviews and observational studies of students learning about concurrency provided insight into these problems. For example, our study of instructors and students in an operating sys-

tems course [15] identified several key difficulties, including:

- Students do not comprehend the space of possible interleavings that arise from the nondeterministic scheduling of threads and are often surprised by the extreme cases, such as when a thread T_1 executes almost to completion before another thread T_2 is scheduled or vice versa.
- Students often conflate the concepts *critical region* and *scheduling policy* to the point that they fail to consider execution sequences in which a thread executing within a critical region is interrupted due to context switching.
- Students have trouble reasoning about why the implementations of synchronization primitives lead to correct synchronization behavior.

We then began to design our saUML sequence diagram notation and evaluate its readability. We evaluated an initial version of this notation via a subjective survey in which students evaluated these diagrams and expressed a feeling that they should be helpful in reasoning about executions of a concurrent program. Specifically, saUML sequence diagrams clarify details that were not discernible from the traditional UML sequence diagrams, such as when a thread enters and leaves a critical region (e.g., a monitor in this study) and which threads are actively running at any given point in a program trace. The survey participants reported the diagrams to be useful in facilitating their understanding of the inherent mechanisms of monitor synchronization.

This paper presents a more refined version of our notation and describes an empirical study to determine whether these diagrams, as opposed to purely textual representations, are able to help students better understand concurrent executions and concurrency concepts as measured by their ability to answer both scenario-based and more general, concept-based questions.

3. The saUML sequence diagram notation

We now briefly describe the characteristics of our notation and discuss its utility through a running example—a monitor-based solution to a simplified version of the bank-account problem, in which two or more threads concurrently access shared bank-account objects. For the example, we modified the standard monitor implementation of a bank-account object to prevent overdrafts on withdrawals.

3.1. Notational specifics

Our saUML sequence diagram notation extends the standard UML 2.0 sequence diagram notation [13] to explicitly represent phenomena related to synchronization. Figure 1 depicts a simple example. The diagram illustrates one possible scenario of interaction among three objects— c_1 , c_2 , and a . Our notation assumes that every class depicted in one of these diagrams must bear one of two stereotypes—*thread* or *monitor*.¹ We distinguish *thread objects* (e.g., c_1 and c_2) using UML’s double-bar convention for depicting active objects. Passive objects (e.g., a) are assumed to be monitors. Thus, the interaction depicted in Figure 1 involves two threads, c_1 and c_2 , and one monitor, a , and illustrates thread c_1 invoking the monitor operation *deposit* on a .

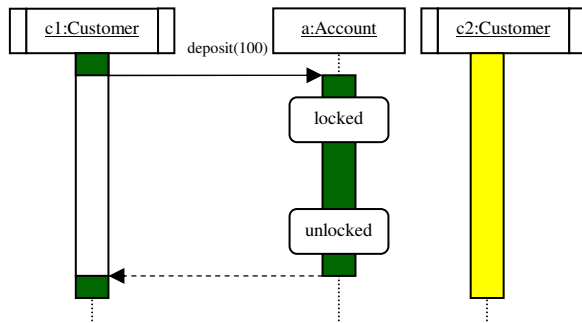


Figure 1. A simple example depicting thread objects and passive objects

We depict synchronization-relevant behavior using an idiomatic combination of UML features and non-standard extensions to represent the *synchronization state* of a monitor and the *scheduling state* of a thread. Every monitor can be in one of two synchronization states—*locked* or *unlocked*—to represent whether a thread is currently executing within it. Changes to synchronization state are depicted using UML *lifeline states* [13, pg 589], i.e, rounded rectangles containing the name of a target state. For example, in Figure 1, the synchronization state of a transitions to *locked* and then later to *unlocked* as a result of thread c_1 executing operation *deposit*. Our notation also allows for the depiction of zero or more problem-specific synchronization states, which are orthogonal to *locked* and *unlocked*. For example, when threads are synchronizing using condition variables, condition changes may be shown explicitly as a change of synchronization state on the lifeline of the monitor. Figure 2 presents

¹ Not shown in this diagram, but would be apparent in the class diagram that defines classes Account and Customer

an example of orthogonal states that indicate the monitor object is locked or unlocked *and* the current account balance is 0 or 100.

Threads may be in one of three scheduling states: *running*, *ready*, or *suspended*. Our notation depicts the current scheduling state of a thread using colored *execution specifications*.² By convention, the thread is running if its most deeply nested execution specification is shaded green, ready if this specification is shaded yellow, and suspended if it shaded red (varying intensities produce shades of gray that are distinguishable in monochrome displays or by color-blind users). Moreover, at any point in time, only the most deeply nested execution specification is shaded. In Figure 1, for example, we see that thread c_1 is *running* for the duration of this interaction, while thread c_2 is *ready*.

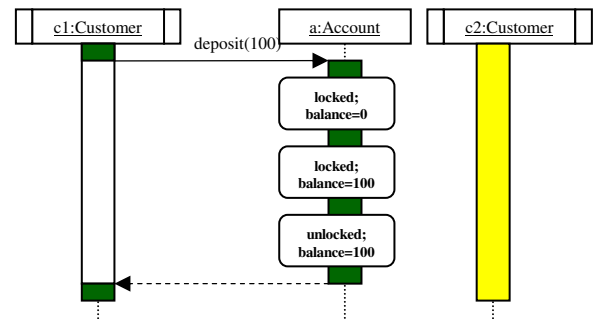


Figure 2. A simple example depicting orthogonal states

3.2. Bank account solution in detail

The bank account problem is stated more formally as follows. Threads represent individual customers, some of whom may share bank accounts. Each customer may deposit or withdraw money from an account; however, to avoid a data race, only one customer may access any account at a time. A customer may execute a deposit at any time, provided that no other thread is currently accessing the account. A withdrawal requires both exclusive access and the condition that the balance be sufficient to permit withdrawal of the requested amount. Figure 3 depicts a Java-like pseudo-code solution to this problem.

Figure 4 presents the implementations of the wait and notifyAll methods using condition variables. Notice that each condition variable maintains a “wait set” onto which threads may be placed to await resumption when some other thread invokes notifyAll for a given condition variable. Wait and notifyAll are methods of class Object; thus “release lock” releases the lock on

² formerly *activations* in UML 1

the monitor and does not affect the condition variable passed in as a parameter. `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 Account extends Object {
    private double balance = 0;
    private CondVar OKtoWithdraw = new CondVar;
    public synchronized void deposit(double amount) {
        balance = balance+amount;
        notifyAll(OKtoWithdraw);
    }
    public synchronized void withdraw(double amount) {
        while (amount > balance)
            wait(OKtoWithdraw);
        balance = balance-amount;
    }
}
```

Figure 3. A Monitor solution to the bank account problem.

```
wait(CondVar cond) {
    put the calling thread on the "wait set" of cond;
    release lock;
    Thread.currentThread.suspend();
    acquire lock;
}

notifyAll(CondVar cond) {
    forall t in wait set of cond
        t.resume()
}
```

Figure 4. The pseudocode for "wait" and "notifyAll"

3.3. A walk-through

Figure 5 uses our saUML sequence diagram notation to present a complex interaction between two customer threads attempting to access a shared account, whose balance is initially \$0. We assume that only those two threads are running and that they are executing on a single physical processor.

The numbers 1-10 shown to the left of the diagram are keyed to the following description:

1. Initially, *c1* is scheduled, as indicated by the green shading (darkest shade of grey on a monochrome display) of its execution specification. Thread *c2* is

ready, as indicated by the yellow (lightest gray) shading of its execution specification.

2. Thread *c1* invokes the `deposit(100)` method and is able to obtain the lock, as indicated by the change in *a*'s synchronization state to `locked`. The orthogonal condition `balance=0` is also depicted.
3. *c1* increases *a*'s account balance by \$100, as indicated by the change in synchronization state with the condition `balance=100` (monitor remains `locked`).
4. A context switch occurs. *c2* now runs (the shading of *c2*'s execution specification changes from yellow to green as that of *c1*'s changes from green to yellow). *c2* invokes `withdraw(150)` and attempts to enter the monitor. However, the monitor lock is held by *c1*, so *c2* suspends (the shading of its execution specification changes to red).
5. *c1* then resumes (shading changes from yellow to green) and calls `notifyAll(OKtoWithdraw)`. Because no thread is suspended on the condition variable `OKtoWithdraw`, this operation does not trigger any changes in scheduling state but releases the lock (`unlocked` state) and returns from `deposit(100)`. Notice, at the point at which *c1* releases the lock, *c2*'s scheduling state changes from `suspended` to `ready` (red to yellow).
6. When another context switch occurs, *c2* is able to run (green shading) and is able to obtain the lock (`locked` state). Because the withdrawal (150) exceeds the balance (100), `wait(OKtoWithdraw)` is invoked. As depicted in the code of Figure 3, *c2* then adds itself to the wait set of `OKtoWithdraw` (not depicted in diagram), releases the lock (`unlocked` state) and suspends itself (shading changes to red).
7. *c1* now resumes execution (green shading). It invokes the `deposit(100)` method, acquires the lock (`locked` state), and increases the `balance` by \$100.
8. *c1* invokes `notifyAll(OKtoWithdraw)`. As a consequence, *c2* (suspended and in the wait set of `OKtoWrite`), now resumes to the `ready` scheduling state (shading changes from red to yellow).
9. *c1* returns from the `notifyAll` method, releases the lock (`unlocked` state) and eventually returns from the invocation of `deposit`.
10. *c2* is then scheduled (green shading). It obtains the monitor (`locked` state), and because the current balance (200) is now greater than the withdrawal amount (150), *c2* eventually completes its transac-

tion and finishes its execution, leaving the account balance at 50 (indicated in the orthogonal state).

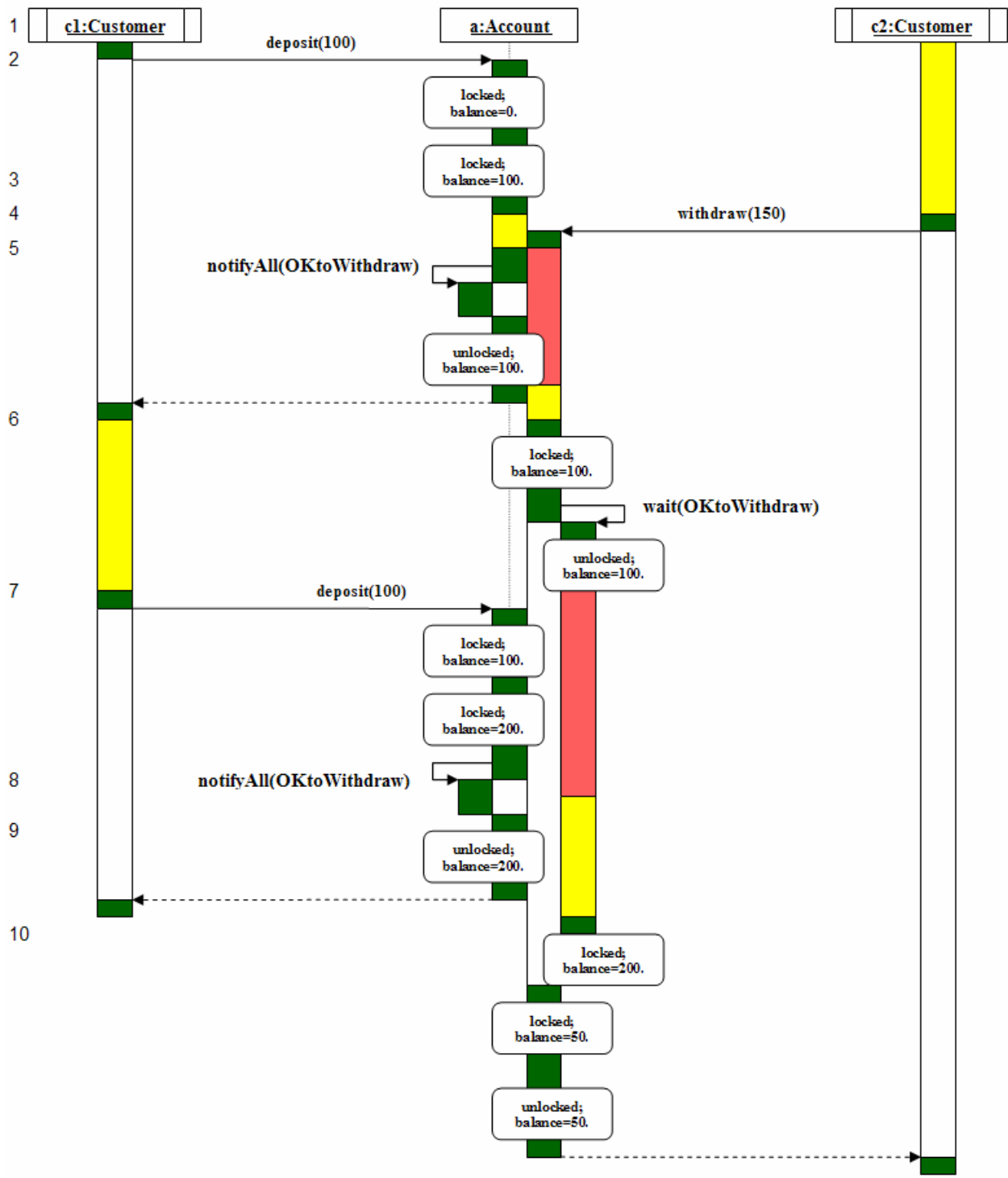


Figure 5. Our saUML sequence diagram notation for the bank account example.

We designed this notation to address the obstacles to student learning uncovered by our surveys and sub-

jective studies [15]. These extensions should help students in comprehension tasks involving the essentials

of thread interleavings and context switches. The diagrams explicitly illustrate thread interactions and the effects of the execution of synchronization primitives by one thread on other threads. Consequently, they reveal more of the underlying dynamics of synchronization, information that is typically hidden from programmers. We expect that this external representation, designed to address some of the difficulties that users encounter in comprehending concurrent programs, will lead to improved performance on comprehension tasks.

4. An objective user study

We conducted an empirical study to examine the benefits of using our saUML sequence diagram notation in conjunction with source code in comprehension tasks for programs that employ concurrency.

4.1. Study Design

We hypothesized that our notation, in combination with a careful selection of motivating examples, has the potential to increase students' performance when answering questions that require them to reason about synchronization behaviors. To evaluate the hypothesis, we adopted a between-subjects, pre-test/post-test design to compare the performance of a text-only group with that of a text-plus-diagram group in answering questions about concurrency. The user study consisted of a teaching session that reviewed concurrency concepts and introduced the monitor construct, a pre-test, another teaching session that reviewed the monitor construct and introduced the readers/writers problem, and a post-test. The data collected from the study showed a statistically significant benefit to the use of our notation.

This study was conducted in an undergraduate operating systems class at the University of Georgia and comprised two 75 minute class sessions. Participants were mostly juniors or seniors who had learned Java in their first two years of study and had recently been taught some basic concurrency concepts in this class. Students were informed that both the pre-test and post-test would be counted as class quizzes. For purposes of class grade reporting, the post-test scores were normalized across groups to ensure fairness.

4.2. First phase of the experiment

In the first 75 minute class session, the experimenter reviewed with the students the basic concepts and terminology in concurrency, including definitions and discussion of threads, context switches, race condi-

tions, atomic operations, critical sections, deadlock, mutual exclusion, etc. A joint bank account example (with no synchronization mechanism) was used to illustrate race conditions. Condition variables and the monitor construct were then introduced, and a monitor solution to a simple rendezvous problem was presented. No diagrams were used.

The pre-test was conducted at the end of the first class session. In the pre-test, we asked both *knowledge level* questions about concurrency concepts and *application level* questions that required students to apply those concepts in interpreting the interactions between two concurrent threads in a single problem. Bloom's taxonomy [2] categorizes the levels of abstraction of questions that commonly occur in educational settings. *Knowledge level* questions are the least abstract, and are characterized by recall of terminology of concepts. *Application level* questions require participants to use methods, concepts or theories in new situations or to solve problems.

The eight knowledge level questions were presented to refresh students' memory about the basic concepts and to make sure that the participants had the fundamental knowledge to understand the terminology used in the subsequent application questions. Of the participants, 83% scored 100% on these eight knowledge level questions and 100% scored 75% or higher, indicating that students had sufficient grasp of the terminology and concepts of interest.

Assume c1 is running within the invocation of deposit(100) and c2 is in the suspended state (it was suspended on the monitor lock).
Question: If c1 releases the monitor lock and leaves the monitor, then:

- c1 changes to *ready* and c2 changes to *running*;
- c1 changes to *suspended* and c2 changes to *running*;
- c1 remains *running* and c2 remains *suspended*;
- c1 remains *running* and c2 changes to *ready*;
- Deadlock occurs.

Figure 6. A sample pre-test scenario and question

Seven application level questions were presented and served as the basis for evaluating student comprehension of the essential synchronization primitives and the behavior of concurrent programs. Figure 6 presents a sample scenario description and question from the pre-test. The question refers to the monitor solution to the bank account solution seen in Figure 3.

Based on their scores on these application-level questions, we divided the students into two groups—a *control* group and a *treatment* (diagram) group—for

the post-test. The division was designed to produce two groups with equal means and standard deviations of pre-test scores. However, only twelve students in the class attended both the pre-test and the post-test sessions. Due to drop-outs between the pre-test and post-test, the actual groups did not have the same mean score on the pre-test. The treatment group received a mean of 3.833 out of 7 with a standard deviation of 0.983 on the pre-test while the control group received a mean score of 4.667 out of 7 with a standard deviation of 1.366. Our analysis methodology took this difference in variances between groups into account.

4.3. Second phase of the experiment

During the next phase of the experiment, the groups attended parallel lecture sessions in different classrooms. Each lecture covered the basic features of monitors and a monitor solution to the bank account problem. Moreover, the PowerPoint slides used to cover this material were exactly the same for both the control and the treatment group. Following this initial lecture, the treatment group was then introduced to our notation and the diagram depicted in Figure 5. These students were then led to “walk through” the event sequence described in Section 3.3. The control group went through the same event sequence as the treatment group, as well as some additional examples to ensure that both groups spent the same amount of time in this second teaching session; however the control group used only the program text (no diagrams).

Afterward, the two groups were brought together and received a brief introduction to the readers/writers problem [5] and one of its monitor solutions (see Figure 7). This problem served as the basis for the majority of the questions and answers involved in the post-tests.

The post-test comprised seven scenarios of program execution traces, each followed by an application level question. These questions were similar to those in the pre-test, except the pre-test questions addressed the simpler bank account problem rather than the more complex readers/writers problem. Figure 8 depicts a sample scenario and its associated question. In addition to these application-level questions, the post-test included three comprehension level questions.

All students then took the post-test. The control group received only textual materials while the treatment group received both the textual materials and diagrams. The diagram in Figure 9 corresponds to the scenario described in Figure 8.

The readers-writers problem is a classic synchronization problem in which two distinct classes of threads exist, reader and writer. Multiple reader threads can be present in the Database simultaneously. However, the writer threads must have exclusive access. That is, no other writer thread, nor any reader thread, may be present in the Database while a given writer thread is present. Note: the reader/writer thread must call *startRead()/startWrite()* to enter the Database and it must call *endRead()/endWrite()* to exit the Database.

```
class Database extends Object {
    private int numReaders = 0;
    private int numWriters = 0;
    private CondVar OKtoRead = new CondVar();
    private CondVar OKtoWrite = new CondVar();
    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 7. A Monitor solution to the readers/writers problem.

Assume the reader thread is in the running state and the writer thread is in the suspended state (we assume the writer thread was previously suspended on *wait(OKtoWrite)*). The reader thread invokes *endRead()* and enters the monitor. It sets the *numReaders* to 0 and issues a *notify(OKtoWrite)*. Note: we assume that only one reader thread and one writer thread are running on the processor.

Question: As a result:

- The reader thread changes to *ready*; the writer thread changes to *running*.
- The reader thread changes to *suspended*; the writer thread changes to *running*.
- The reader thread remains *running*; the writer thread remains *suspended*.
- The reader thread remains *running*; the writer thread changes to *ready*.
- Deadlock occurs.

Figure 8. Scenario and question from the post-test.

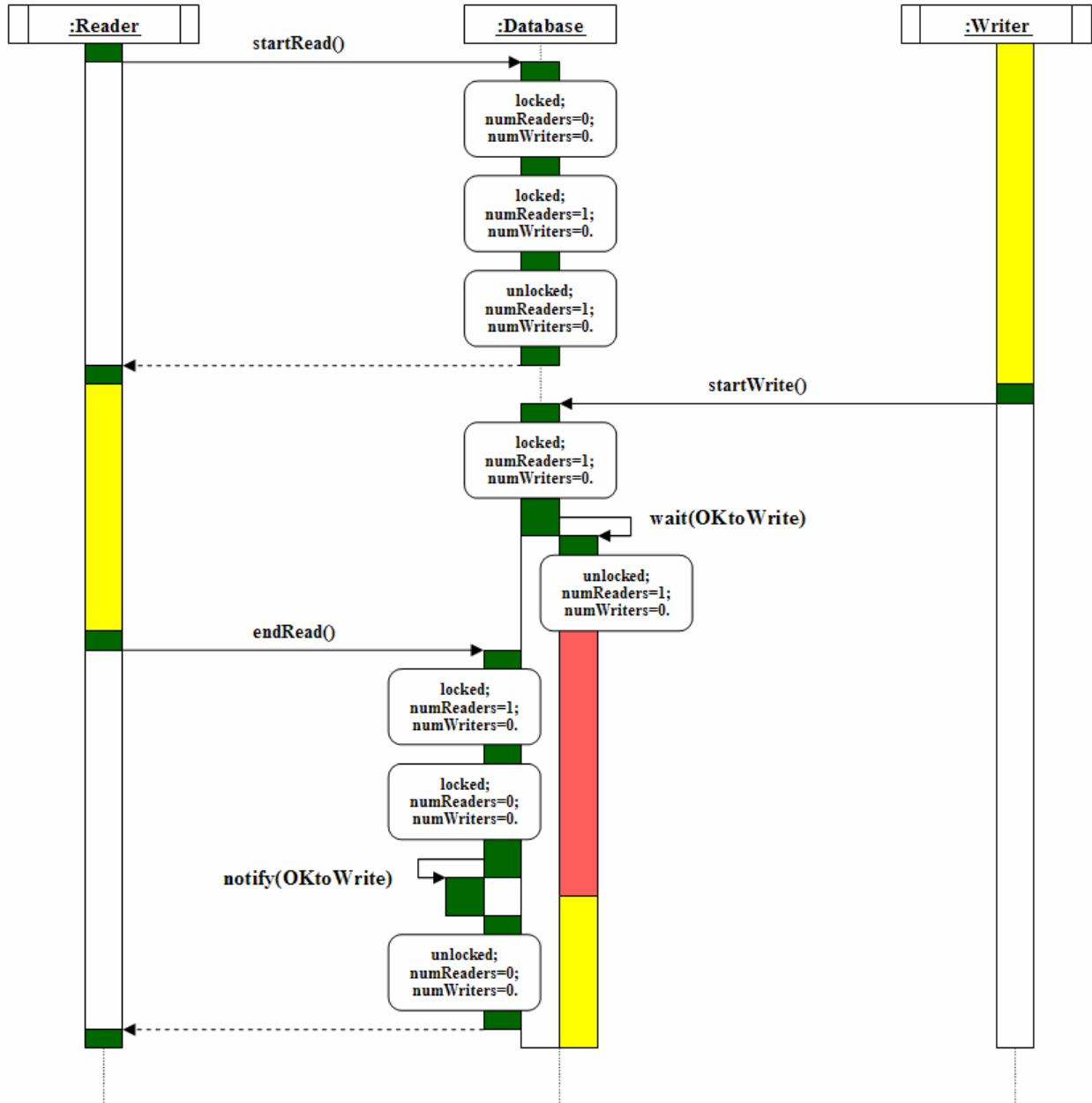


Figure 9. A sample diagram appeared in the post-test, depicting an execution of the readers/writers program.

4.4. Results and analysis

Table 1 summarizes the results of the study. The mean score of the control group dropped from 4.667 (67%) on the pre-test to 3.667 (52%) on the post-test, while the mean score of the treatment group rose from 3.833 (55%) to 4.833 (69%). We believe the drop in performance on the part of the control group was due to the use of a more complex problem (readers/writers)

on the post-test than what was used on the pre-test. Structurally, the pre-test and post-tests were equivalent, as each contained only seven application-level questions with scenario descriptions of comparable length.

To measure the effect that our notation had on the subjects' ability to perform problem-solving tasks, we compared the changes in scores from pre-test to post-test (post-test scores minus pre-test scores). By applying a two-tailed heteroscedastic (does not assume equal

variance) t-test to the improvement matrix, we obtained a p-value of 0.027. This result indicates a statistically significant difference between the two groups, with the treatment group outperforming the control group.

Table 1. Statistical results

| | Control Group | Treatment Group |
|-----------------------------------------------------|---------------|-----------------|
| Pre-test Mean / Stdev | 4.667 / 1.366 | 3.833 / 0.983 |
| Post-test Mean/Stdev | 3.667 / 1.506 | 4.833 / 0.753 |
| Improvement (Post-test - Pre-test) Mean/Stdev | -1 / 1.265 | 1 / 1.414 |

The post-test also included three comprehension level questions. In contrast to the application-level questions, these questions were more general in nature and did not have a corresponding diagram. Nevertheless, the treatment group outperformed the control group in that the mean score of the treatment group was 2.667 out of 3 while the mean score of the control group was 2.167. We postulate that our notation, which also visualizes the low-level implementation details of some of the synchronization primitives, may aid students in achieving the comprehension-level objectives in learning about concurrency. However, due perhaps to the limited group size and the limited number of comprehension level questions presented in the study, the result was not statistically significant.

In summary, our saUML sequence diagram notation resulted in a statistically significant benefit in answering application-level questions. A beneficial trend was observed for comprehension-level questions.

5. Conclusions and future work

In this paper, we have presented the details of our saUML sequence diagram notation, which extends the UML 2.0 sequence diagram to depict thread scheduling/synchronization phenomena. One must reason about these phenomena when performing comprehension tasks involving concurrent programs. We believe that well-designed external representations can aid in these tasks, perhaps enabling a larger population of developers to contribute to this end. Using empirical observation, we have shown the representation presented here to be beneficial as compared to a text-only presentation. Future studies will allow us to compare the efficacy of this representation with other graphical notations. In fact, an experiment comparing our notation versus the original UML sequence diagram notation is in progress.

Meanwhile, we are formalizing our saUML sequence diagram notation in the forms of UML 2.0 profiles [13]. Such profiles tailor our notation for modeling concurrent software and make it possible to introduce tool support for our notation.

Our ultimate goal is to develop external representations that aid developers in the problem-solving tasks that attend to the design, verification, and maintenance of concurrent software. Program comprehension plays a major role in each of these activities, especially verification and maintenance. We believe that many of the problems that complicate learning about the proper use of concurrency and synchronization also contribute to the complexity of comprehension tasks in this domain, and to this end, we have begun to investigate alternative representations that improve comprehension during learning. This paper describes one study in our continuing work toward this goal.

6. Acknowledgements

The material is based upon work supported by the National Science Foundation under Grants EIA-0000433 and IIS-0308063. Additional partial support for Stirewalt provided by ONR grant N00014-01-1-0744. Our thanks to Maria Hybinette for her collaboration and support in the user study conducted in her class.

7. References

- [1] R. Behrends and R. E. K. Stirewalt, The Universe Model: An approach for improving the modularity and reliability of concurrent programs, *In Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2000, 20-29.
- [2] B. S. Bloom, *Taxonomy of Educational Objectives, Handbook 1: Cognitive Domain*, David McKay, New York, 1956.
- [3] S. Carr, J. Mayo and C.-K. Shene, ThreadMentor: a pedagogical tool for multithreaded programming, *Journal on Educational Resources in Computing (JERIC)*, 3, 1, (2003).
- [4] S.-E. Choi and E. C. Lewis, A Study of Common Pitfalls in Simple Multi-Threaded Programs, *In Proceedings of the ThirtyFirst ACM SIGCSE Technical Symposium on Computer Science Education*, 2000, 325-329.
- [5] P. J. Courtois, F. Heymans and D. L. Parnas, Concurrent control with "readers" and "writers", *Communications of the ACM*, 14, 10, (1971), 667 - 668.

- [6] C. Exton and M. Kolling, Concurrency, objects and visualization, *Proceedings of the Australasian conference on Computing education*, ACM Press, Melbourne, Australia, 2000, 109-115.
- [7] C. Hughes, J. Buckley, C. Exton and D. O'Carroll, Towards a Framework for Characterising Concurrent Comprehension, *Computer Science Education*, 15, 1, (2005), 7-24.
- [8] Y. B.-D. Kolikant, Learning concurrency: evolution of students' understanding of synchronization, *International Journal of Human-Computer Studies*, 60, 2, (2004), 243-268.
- [9] J. Kramer, Abstraction - the key to computing, *Communications of the ACM*, 50, 4, (April, 2007).
- [10] H. Leroux and C. Exton, Visualising the execution of concurrent object oriented programs dynamically using UML, *The 9th International Conference in on Computer Graphics, Visualization and Computer Vision*, Czech Republic, 2001.
- [11] C. E. McDowell and D. P. Helmbold, Debugging concurrent programs *ACM Computing Surveys (CSUR)*, 21, 4, (1989), 593-622.
- [12] R. E. K. Stirewalt, R. Behrends and L. K. Dillon, Safe and reliable use of concurrency in multithreaded shared memory systems, *In Proc. of the 29th Annual IEEE/NASA Software Engineering Workshop*, 2005.
- [13] J. Rumbaugh, I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 2004.
- [14] 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*, 14, 1, (1998), 12-24.
- [15] S. Xie, E. Kraemer and R. E. K. Stirewalt, Design and Evaluation of a Diagrammatic Notation to Aid in the Understanding of Concurrency Concepts, *The International Conference on Software Engineering*, Minneapolis, 2007.
- [16] J. Zhang and D. A. Norman, Representations in Distributed Cognitive Tasks, *Cognitive Science*, 18, (1994), 87-122.