Design and Evaluation of Extensions to UML Sequence Diagrams for Modeling Multithreaded Interactions

Abstract

Learning about concurrency and synchronization is difficult for novices. Our research seeks to support and improve the teaching and learning of concurrency concepts and to improve comprehension of the intricacies of multiple thread interactions. This paper describes a series of empirical studies in the first phase of our research. We began by conducting a comparative study to empirically evaluate the usability by novices of the existing variants of the UML sequence diagram notation in solving comprehension tasks involving multiple thread interactions. The results implied that a deliberately designed variant of this notation may provide better support for reasoning about concurrent behavior. We then investigated the factors that complicate learning, with the idea that the same complexities would also complicate comprehension tasks. In order to understand the practical difficulties novices encounter in learning about concurrency, we conducted an instructor interview and an observational study. These investigations guided us in determining the desirable properties of a new notation. We then designed *synchronization-adorned UML* (saUML) sequence diagrams, which extend UML sequence diagrams with those properties. Finally, we performed four empirical studies to evaluate the usability and efficacy of saUML. Through these empirical studies, we were able to validate the benefits of saUML in enhancing novices' understanding of programs with different levels of synchronization complexity.

Keywords: UML, Empirical Evaluation, Concurrency and Synchronization.

CR Categories: D.1.3 [Software]: Programming Techniques—Concurrent Programming; D.2.5 [Software]: Software Engineering—Testing and Debugging

1 Introduction

Multi-threaded software is difficult for programmers, especially novices, to understand [Choi and Lewis 2000; Ben-David Kolikant 2004; Kramer 2007]. Comprehension is complicated by many factors, including:

1. the delocalized nature of synchronization logic, which is often tangled with the "business logic" of the program,
2. the nondeterministic nature of thread scheduling, which gives rise to a large space of potential execution traces, and
3. the need to understand how one thread synchronizes with another through low-level primitives, which synchronize threads indirectly by modifying OS-level data structures such as condition queues and mutex locks.

Our research seeks to support and improve the teaching and learning of concurrency concepts and to investigate the characteristics of visual representations that improve programmer comprehension of the intricacies of multiple thread interactions. This paper synthesizes the results of multiple studies in which the visual representation under study is a variant of the UML 2.0 sequence diagram notation.

We chose to investigate UML sequence diagrams for several reasons. First, UML is widely accepted in both academia and industry. Of its many notations for behavioral modeling, sequence diagrams and communication diagrams[1] are best suited for depicting concrete scenarios of interaction, such as are used by instructors in teaching concurrency concepts and by developers in diagnosing concurrency-related failures. Of these candidates, we prefer sequence diagrams because their visual features (e.g., object lifelines and activations) are easily adorned with state information, which is awkward to depict in communication diagrams. A potential drawback of sequence diagrams is that each diagram depicts only a single scenario. However, we have found that a small collection of well-chosen sequence diagrams often suffices to explain the essence of a concurrent design. Finally, the sequence diagram notation in particular is widely used in practice.

[1] known as *collaboration* diagrams in UML 1.X

The UML sequence diagram notation provides several features that are useful in modeling interactions among concurrent agents—notably, the ability to designate *active objects* and to designate *execution specifications*, which depict the lifetime of an activation of a method on an object by some caller. By modeling each thread as an active object, a sequence diagram depicts an abstract view of the dynamics of synchronization, showing the time ordering of interactions between threads in a single program trace. Although earlier versions of UML sequence diagrams attempted to model thread-level synchronization [Schader and Korthaus 1998; Mehner and Wagner 2000], the current notation provides little support for doing so [Ober and Stan 1999; Stevens 2003].[2]

Our first study was a comparative study that evaluated several variants of the UML sequence diagram notation for usability by novices on comprehension tasks involving multi-threaded programs that synchronize using mutex locks. The results of this study implied that a deliberately designed variant might provide better support for reasoning about the behavior of concurrent programs than do existing notations. This finding prompted us to investigate factors that complicate learning, expecting that the same factors would also complicate comprehension.

To understand the practical difficulties novices encounter in learning about concurrency, we conducted an instructor interview and an observational study [Xie et al. 2007(b)]. Guided by the results from the comparative study, the interview, and the observational study, we designed *synchronization-adorned UML* (saUML) sequence diagrams to extend UML sequence diagrams with features to address these difficulties. Using a combination of color and textual adornments, saUML extends UML 2.0 sequence diagrams to depict the execution status of threads (i.e., blocked or able to run) and the status of mutex locks and counter variables. By making these concepts explicit, saUML diagrams expose many of the otherwise invisible details at play during thread synchronization.

We then performed four empirical studies to evaluate the usability and efficacy of saUML. The first of these, a subjective survey [Xie et al. 2007(b)], assessed readability and user acceptance of saUML. The other three studies were controlled, objective studies. The first objective study investigated how saUML sequence diagrams, as compared with purely textual representations, aid novice programmers in compre-

[2]Note: Although these papers were published prior to the drafting of UML 2.0, their conclusions remain largely true today. One caveat is that UML 2.0 now provides a means for marking a sequence diagram as a critical region [Rumbaugh et al. 2004].

hension tasks [Xie et al. 2007(a)]. The second and third objective studies [Xie et al. 2008] compared user performance on questions dealing with concurrency and synchronization when given either saUML sequence diagrams or standard UML sequence diagrams.

The remainder of the paper is organized as follows. The next section summarizes related work on support for learning about concurrency and synchronization and on studies that evaluate the UML notation. In Section 3 and Section 4 we describe the studies that served as the basis for the development and refinement of the saUML notation, and then present details of the saUML notation in Section 5. Next, Section 6 and Section 7 describe our empirical studies of the benefits of saUML on comprehension tasks. Finally, we present conclusions in Section 8.

2 Related Work

2.1 Learning and Teaching Concurrency

Several studies have looked at how students learn about concurrency and the kinds of problems that attend to this learning. Choi and Lewis [Choi and Lewis 2000] conducted a detailed study of student programs involving concurrency and synchronization and found that over 30% contained serious concurrency-related design flaws, even though the programs produced correct output when tested against sample test cases. A controlled experiment conducted by Ben-David Kolikant [Ben-David Kolikant 2004] showed that novice students often develop pattern-based techniques to successfully solve synchronization problems and avoid dealing with the dynamics of the synchronization mechanisms. However, the experiment also showed that these novice students, perhaps as a result of their reliance on those pattern-based approaches, have trouble in solving non-familiar-pattern synchronization problems. It seems that students cannot reason about thread interaction in novel situations (i.e., situations that do not conform to a standard pattern) without a deep understanding of how the effects of low-level primitives on operating-system states and data structures indirectly affect thread synchronization.

A few teaching tools have been developed to facilitate the teaching of concurrency at a higher level of abstraction. For example, Higginbotham and Morelli [Higginbotham and Morelli 1991] designed a con-

current programming interface based on the use of semaphores to correspond closely to some of the classical semaphore programming examples. Carr, et al. [Carr et al. 2003] designed the ThreadMentor system, which dynamically displays the current state of each thread (i.e., running, blocking, etc.), and which threads, if any, are waiting on each mutex, semaphore, or condition variable. ThreadMentor also produces a static visualization of the history of each thread, which shows how each thread's state changed over time and how the threads synchronized (e.g., which thread's invocation of `signal` awoke which waiting thread). These visualization tools and representations, while useful in providing a high-level view of how synchronization primitives (locks, semaphores and monitors) can be applied in concurrent programming, are not capable of revealing the underlying mechanisms of those synchronization primitives. That is, they are not helpful in explaining why and how those synchronization primitives achieve their goals, nor in providing students with insight that will allow them to apply their knowledge in novel situations that deal with thread synchronization through the synchronization primitives.

2.2 Visualizing Thread Interactions

Approaches to the visualization of thread interactions can be classified as either static or dynamic. Our work is concerned with static visualizations, such as UML sequence diagrams. Mehner [Mehner 2002; Mehner and Wagner 2000] takes a static approach, extending the syntax for UML sequence diagrams with shading conventions on execution specifications to indicate when threads are actively executing. SaUML borrows from and extends these conventions. In addition to shading, Mehner's extension includes calls to a distinguished *synchronize* operation, which is invoked to lock an object. Calls to Java `synchronized` methods are modeled using execution specifications that begin with a call of the form $synchronize(this)$. Likewise, entry into a Java synchronized block on some object o is modeled by a call of the form $synchronize(o)$. Because calls to this distinguished operation may block, this extension depicts the blocking of threads, which makes deadlocks easier to recognize. However, Mehner's extension abstracts away subtle details, such as when locks are released during condition synchronization. These details are essential for a deep understanding of the intricacies of thread interactions. For instance, when a thread invokes `wait` on a condition variable, the thread releases the lock associated with that condition

variable before suspending. The release is not shown explicitly in the code because it happens inside the body of the `wait` primitive. By abstracting away such implementation details, high-level visualizations may fail to dispel common student misunderstandings, such as that a thread continues to hold the lock while waiting.

Newman and colleagues [Newman et al. 2001] propose two new diagramming notations to statically visualize concurrency-related design decisions that are not easily derived through code inspection. Their *regional state hierarchy diagram* extends the UML class diagram to depict the structure of lock-state associations. The notation identifies the shared state within a set of classes and describes how that state is protected. Their *method concurrency diagram* extends a call graph to show which methods invoke operations on which mutexes and condition variables, which mutex protects each condition variable, and which data each lock protects. Although Newman and colleagues' diagrams do not model thread interactions explicitly, they provide supporting information that could be useful in conjunction with, for instance, a sequence diagram.

The literature contains numerous dynamic visualizations of thread interactions. Traditional parallel debuggers (e.g., [McDowell and Helmbold 1989]) provide a rudimentary visualization by displaying a debugger window for each thread. Other visualization tools show the status of various properties as a multithreaded program executes. Leroux and colleagues [Leroux et al. 2003] developed a tool, Jacot, that dynamically elaborates a standard UML sequence diagram, and as the diagram grows, the tool also displays the current state of each thread.

2.3 Evaluations of UML Diagrams

Researchers have studied the usability of various types of UML diagrams. Kutar and colleagues [Kutar et al. 2002] compared the impact of collaboration and sequence diagrams on comprehension and found no significant difference between them in performance. In a later study, Swan and colleagues [Swan et al. 2005] found no difference among participants familiar with both types of diagrams, but they observed that sequence diagrams were easier to use by those who were unfamiliar with either type. Torchiano [Torchiano 2004] compared the use of class diagrams alone to the use of class diagrams plus object diagrams as aids

in answering questions about simple programs. His results suggest that object diagrams have a positive effect when the class diagrams are complex—for example, containing many classes and having cyclical associations (e.g., as in the Composite design pattern [Gamma et al. 1995]).

Other researchers have looked at the impact of UML-based documentation on the ability to comprehend a software system. Tilley and Huang [Tilley and Huang 2003] conducted a study in which UML experts were given a series of UML diagrams (e.g., use-case, class, and sequence diagrams) depicting a software system and asked to answer questions about the system. The study revealed problems with layout, lack of support for representation of domain knowledge, and unclear specifications of syntax and semantics of some advanced modeling features as factors that limited support for program understanding. More recently, Arisholm and colleagues used students proficient in object-oriented programming and UML modeling to evaluate the impact of using UML on the correctness of and effort required for certain software maintenance tasks [Arisholm et al. 2006]. They found that the use of UML reduced the time required to make code changes but that these savings are largely negated by the time required to modify the diagrams.

Still other researchers have explored variations in the visual presentation of UML diagrams. Purchase and colleagues [Purchase et al. 2001] evaluated the effect of various aesthetic attributes (i.e., edge length, node distribution, and other layout characteristics) on viewers' comprehension of UML diagrams. Kuzniarz and colleagues [Kuzniarz et al. 2004] studied the effects of UML stereotypes (i.e., graphical icons) on program comprehension, as measured by performance and time to completion. They found that use of the stereotypes both increased the rate of correct responses and decreased the time to completion.

3 Comparative study of variations of UML sequence diagrams

Our initial study compared standard UML 2.0 sequence diagrams [Rumbaugh et al. 2004] and two variants of sequence diagrams for their ability to help users evaluate concurrency concerns. The two variants use shading to represent the execution status of threads, but interpret shading differently. The first variant is due to Mehner and Wagner [Mehner and Wagner 2000]; for simplicity, we refer to it in the sequel as "Mehner's variant." The second was developed independently by the third author to use in teaching concurrency concepts to undergraduate students at Michigan State University; we refer to it in the sequel

as “Stirewalt’s variant.”

A sequence diagram depicts a specific interaction among a collection of objects over time. The interaction is expressed as an exchange of messages, each of which emanates from one object and activates (or returns from activation of) another. Fig. 1 (a) shows a standard UML sequence diagram. Each box in the top row represents one of the objects being modeled. The underlined label on an object box provides an (optional) object name (preceding the colon) and indicates the object’s type (following the colon). In Fig. 1, `s1` and `s2` designate objects of type `Stylus`. The two boxes labeled `:Widget` represent anonymous objects of type `Widget`. In addition, objects, such as `s1` and `s2`, whose boxes are marked with additional vertical bars, are said to be *active* rather than *passive*. Active objects host their own thread of control.

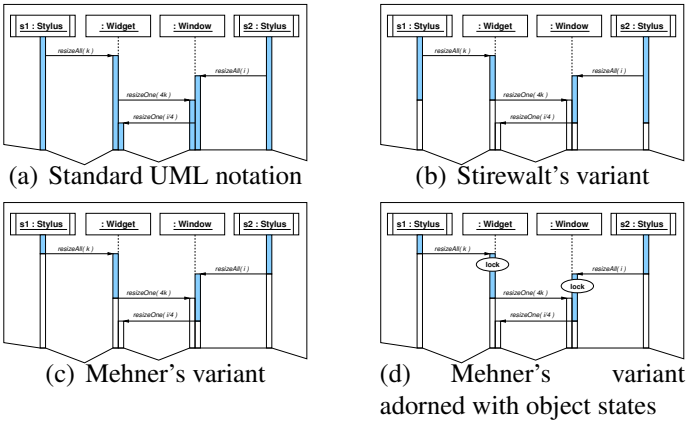
The vertical dimension represents time, which increases from top to bottom. A horizontal arrow depicts a message exchange, which, in our examples, will always signify either the invocation of or return from a method on some object. The arrow emanates from the object that sent the message and points to the object that received the message. Below an object box, a dashed line, called a *lifeline*, shows the period of time during which the object exists. An *execution specification*, drawn on top of or adjacent to an object’s lifeline, depicts the activation of a method that the thread has invoked on the object and that has not yet returned. In the case of an active object, the thread invokes a special method, called the object’s *run method*.[3]

In standard UML, an execution specification is always shaded. In the Mehner and Stirewalt variants, shading provides information about the execution status of threads. Specifically, the Mehner variant leaves sections of an execution specification white (unshaded) when the host thread is either hosting a more recent execution specification or blocked waiting to acquire a lock. In Fig. 1(c), for example, the execution specification on `s1`’s lifeline starts out shaded and becomes white when the thread calls `resizeAll(k)` on the first (left) `Widget` object. Similarly, the execution specification representing the activation of `resizeAll` created by this call starts out shaded and becomes white when the thread (executing in the first `Widget` object) calls `resizeOne(4k)` on the second (right) `Widget` object. However, the execution specification representing the activation of `resizeOne` created by this call is never shaded

[3]Because our experiments have not yet looked at understanding the semantics of object initialization and termination, we depict scenarios that start after the run methods of active objects have been invoked.

because the thread blocks on this call. In contrast to the Mehner variant, the Stirewalt variant leaves sections of an execution specification white only when the thread hosting the activation is blocked waiting to acquire a lock. Thus, in Fig. 1(b), the execution specification on `s1`'s lifeline does not turn white until the thread blocks.

UML 2.0 *object states* are used to depict meaningful changes in the abstract state of an object during an interaction. Object states are depicted using ovals with embedded text, e.g., `lock` in Fig. 1(d), and located along the object's lifeline at the point in time in which the object enters the named state. Using this notation, execution specifications may be annotated to indicate the entry of the object into a distinguished object state, of which we were interested in two: locked and unlocked. An object enters state locked (depicted `lock` in Fig. 1(d)) when a thread acquires the lock associated with that object and enters state `unlocked` when the thread holding the lock releases it.



(a) Standard UML notation (b) Stirewalt's variant

(c) Mehner's variant (d) Mehner's variant adorned with object states

Figure 1: *Variations of the UML sequence diagram notation depicting a deadlock scenario in which two users concurrently manipulate a multi-user editor.*

Our comparative study sought to measure the relative benefits of each variant of the UML sequence diagram (standard, Mehner, and Stirewalt) both with and without the use of object states. The study involved twenty four participants, of which eight were undergraduate and sixteen were graduate students, all Computer Science students from the University of Georgia. We randomly divided all participants into six groups of four participants each. Table 1 shows the experimental design.

Participants in each group completed a questionnaire involving interactions among objects in a multi-user graphical object editor, which in this case comprises a window that contains a resizable rectangular widget.

Table 1: *3×2 Factorial Experimental Design - Number of subjects per group*

	standard UML	Mehner's	Stirewalt's
With object states	4	4	4
Without object states	4	4	4

The editor was designed so that the proportion of the window occupied by the widget remains constant. That is, if one user uses stylus `s1` to resize the widget, a resize of the containing window will be triggered. Likewise, if another user uses stylus `s2` to resize the window, a resize of the contained widget will be triggered.

We created a version of the questionnaire for each combination of shading variant (standard, Mehner, or Stirewalt) and the use (or lack of use) of object states to depict the locking status of passive objects for a total of six versions. Except for these conventions, the six versions were identical. The questions involved three distinct kinds of interaction:

- a non-blocking, non-deadlock interaction,
- a blocking but non-deadlocking interaction, and
- a deadlocking interaction,

all involving the same collection of objects. For each group, each of the three interactions was depicted by the assigned variant of the UML sequence diagrams.

Figure 1 depicts four of the six versions of a deadlocking interaction that occurs when two threads (represented by `s1` and `s2`) are each holding a lock on a shared resource (widget or window) and waiting for another to release the lock. Note that Fig. 1(d) presents an example in which Mehner's variant is adorned with object states. The standard UML variant and Stirewalt's variant can also be adorned similarly with object states. Subjects were presented with a collection of three diagrams and were asked to answer a set of fourteen questios.

The questions were of three distinct kinds:

1. When/where does a thread obtain/release a lock on a particular object?

2. When/where does a given thread block attempting to obtain such a lock?

3. Does a given interaction illustrate a synchronization flaw (e.g., a data race or a deadlock)? If so, the subjects were asked to answer further questions relevant to the specific flaw.

Our collected empirical results showed significant differences in user performance across those variants only for questions related to lock acquisition and release($p=0.024$, a two-tailed, unequal variance t-test).

Not surprisingly, users from groups dealing with diagrams that explicitly depicted object states were much more accurate in answering these kinds of questions. Differences in shading conventions across the three variants were not found to significantly affect performance. However, the study was small and involved only simple forms of synchronization (e.g., no use of condition synchronization and only a small number of collaborating objects). The results of this pilot study suggest that UML sequence diagram adornment can improve user performance for certain kinds of comprehension problems involving behaviors of multi-threaded programs. These results leave open the potential benefits of shading conventions. Our subsequent studies therefore focused on richer shading conventions than those explored in this pilot study and on a richer corpus of synchronization problems and levels of synchronization complexity.

4 Instructor Interview and Observational Study

Motivated by findings of our pilot study, we next conducted instructor interviews and an observational study [Xie et al. 2007(b)] to identify difficulties that students commonly experience when learning about concurrency. We individually interviewed three instructors who teach courses that deal with concurrency concepts at the undergraduate and graduate levels. Each professor was given a list of key concurrency concepts and related example problems. The professor was then asked to identify the concepts students found difficult and to describe how he or she attempted to help students better understand these concepts.

To confirm the common difficulties noted by instructors in the interviews and identify additional difficulties, we conducted an unobtrusive observational study in a graduate-level operating systems class. During the five weeks of class sessions that dealt with concurrency and synchronization, we observed student behavior in the class and took notes on the instructor's presentation, the students' questions, and student

responses to the instructor's questions.

Problems observed in this study include:

1. The instructor often used ad-hoc sketches to describe concurrent program executions. However, students frequently had trouble both copying detailed sketches and recording their explanations for later review of the reasoning behind the sketches.

2. Students had trouble envisioning the large space of potential execution sequences that could arise due to different thread schedules and comprehending consequences of different schedules.

3. Students often conflated the concept of blocking on entry to a critical region with that of preemption by the scheduler. A common mistake was thus to fail to consider execution sequences in which a thread within a critical region is interrupted due to context switching.

4. Consistent with the findings of Ben-David Kolikant [Ben-David Kolikant 2004], students were often confused by questions related to the effects of minor modifications of standard idioms for concurrency control or of implementations of synchronization primitives (e.g. monitors). We consider this confusion indicative of trouble reasoning about how synchronization primitives "work."

5. Students frequently had trouble choosing an appropriate idiom to meet specific synchronization goals and/or correctly instantiating available synchronization primitives. This observation is consistent with findings by Shene and Carr [Shene and Carr 1998].

5 SaUML Sequence Diagrams

From the above difficulties we synthesized the following list of desirable properties of representations to support students in learning how to use concurrency and synchronization primitives:

- Depict states of OS resources (e.g., thread scheduling states, lock states, etc.) explicitly so that students can understand how threads synchronize with one another through interaction with low-level primitives.

- Depict application-specific synchronization states and conditions, e.g., values of counter variables, whether a bounded buffer is empty, full, etc.
- Depict how the invocations of operations affect those states.
- Support "at a glance" detection of global synchronization properties, e.g. deadlock, safety violation.

We designed saUML to address these difficulties. In saUML, visual adornments depict the execution status of threads, states of simple OS resources (e.g., mutex locks) and application-specific synchronization states and conditions. Other resources, such as condition variables, are depicted as first-class shared objects, distinct from the conditions that these resources are used to enforce. [4] By virtue of its ability to depict these conditions and the states of OS resources, the diagrams clearly connect changes to these conditions and states that result from the invocation of operations. SaUML also supports the "at a glance" detection of some global synchronization phenomena—specifically deadlock. Whether it can be used to quickly detect violations of safety and liveness properties is a topic of future research that is beyond the scope of this paper.

saUML is designed to depict programs written in an architectural style in which multiple threads vie for exclusive access to one or more shared objects. Following the conventions and terminology of Magee and Kramer [Magee and Kramer 2007], we assume a shared object can behave as a *monitor*, i.e., an object that guarantees mutually exclusive access to its critical data. Monitors are usually implemented by means of a mutex lock, which is acquired prior to executing the body of a method and released on return. Moreover, in real designs, an object may not be a *strict monitor*, which is the case if some but not all of its operations guarantee mutual exclusion. Here, we use the term monitor to include such objects and refer to operations that guarantee mutual exclusion as *monitor operations* and to others as non-monitor operations. Thus, a saUML diagram depicts a scenario of interaction among a collection of threads and monitors in this sense.

We now briefly describe the characteristics of our notation and discuss its utility through a running example—a monitor-based solution to a simplified *readers–writer problem*, whereby clients of two types

[4] This distinction between the condition variable, which is an OS resource, and the conditional synchronization, which the resource is employed to implement, is subtle and often difficult for novices to grasp. saUML diagrams attempt to address these difficulties by separating and explicitly representing and connecting these distinct but related concepts.

(reader and writer) attempt to access a shared database object(Fig. 2).

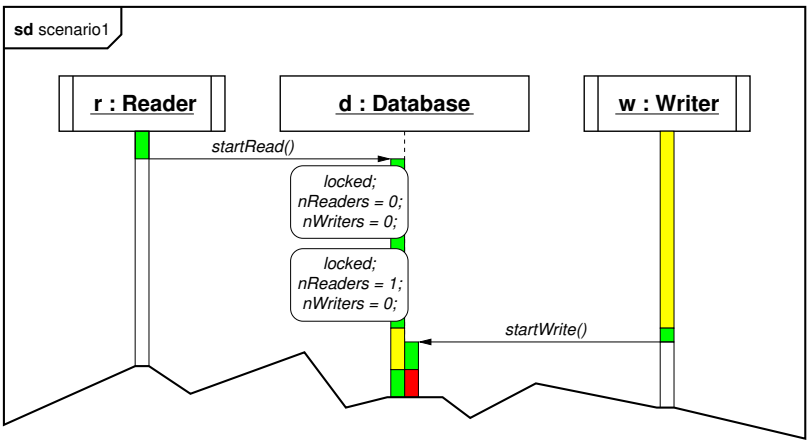


Figure 2: *Sample saUML sequence diagram.*

The saUML extensions include two new features: 1) a refinement of Mehner's variant in which execution specifications are colored to represent the scheduling states of threads and 2) UML object states to indicate the synchronization states of monitors. As in standard UML sequence diagrams, active objects (those with their own thread of control) are designated using a double bar on the object box. Fig. 2 depicts a scenario involving two threads—a reader thread r and a writer thread w. By convention, the state of a thread at any point in time is running if its most recent execution specification is shaded green, ready if this specification is shaded yellow, and blocked if it is shaded red. [5] Moreover, at all times and for each thread, only the most recent execution specification associated with that thread is shaded.[6] In Fig. 2, for instance, thread r is initially running, while thread w is initially ready.

The synchronization states of a monitor indicate whether the mutex used to guard access to monitor operations is *locked* or *unlocked*. Synchronization states may also associate values to problem-specific counter variables and conditions. Changes to synchronization state are depicted using UML object states [Rumbaugh et al. 2004, pg 589], i.e, rounded rectangles containing assertions that describe the synchronization state. For instance, in Fig. 2, the database d transitions to a state in which d is `locked` as a result of thread r executing operation `startRead`. At this point, any other thread that invokes a monitor operation on d will block until such time as the executing thread unlocks d. Additionally, counter variables `nReaders`

[5] Varying intensities produce shades of gray that are distinguishable in monochrome displays or by color-blind users.

[6] The most recent execution specification associated with a thread models the activation at the top of that thread's run-time stack.

and `nWriters` are both zero in this state; these counter variables record the number of reader threads and writer threads, respectively, that are currently "in" (i.e., authorized to access) the database. Thus, when threads are synchronizing using condition variables, condition changes are shown explicitly as state changes on the lifeline of the monitor.

With these conventions in hand, a programmer would read the scenario depicted in Fig. 2 as follows. A reader thread and a writer thread are both active when the program starts. The reader thread is scheduled first. It invokes a monitor operation `startRead` and obtains the monitor lock on d. After the reader thread sets the counter `nReaders` to one, a context switch occurs. The writer thread is then scheduled. It invokes a monitor operation `startWrite` and tries to obtain the monitor lock. However, because the lock is held by the reader thread, the writer thread blocks and the reader thread resumes.

While saUML resembles Mehner's notation, it differs in that it does not conflate thread state *ready* with thread state *running*, nor does it conflate blocking due to synchronization with the activation of a nested procedure. SaUML also represents condition variables as first-class objects, which permits an explict depiction of the effects of `wait` and `notify` on lock states and thread synchronization. These fine distinctions made by saUML address many of the knowledge gaps that novices often stumble on.

6 Subjective Study

Next, we conducted a subjective user study [Xie et al. 2007(b)] of our notation using five graduate students in the Computer Science Department at the University of Georgia, all of whom had recently been taught about semaphores and monitors.

This subjective study consisted of a short lecture and a survey. We first introduced saUML sequence diagrams to the subjects and then, using the readers-writers example, demonstrated how to apply saUML sequence diagrams to model interactions between multiple concurrent threads. Finally, we asked the subjects to fill out a short survey. This pilot study was designed to be exploratory and formative in nature.

Participants were asked to rate, on a scale from 1 (strongly disagree) to 5 (strongly agree), how well the saUML diagram:

- clarifies when a thread enters and exits a monitor routine—average rating: 4.4;
- clarifies when and which threads are actively running on the processor at any given time, assuming threads share a single processor—average rating: 4.2;
- illustrates the interactions between threads in a single program trace—average rating: 4.3;
- facilitates understanding of the inherent mechanisms of monitors—average rating: 4.0;

Other questions asked participants about their familiarity with UML and asked them to suggest additional refinements to the diagrams. Overall, the collected results were encouraging in that all participants either agreed or strongly agreed that saUML helped clarify how the execution unfolds.

7 Objective User Studies

Study	Notations	Synch. features
saUML+	saUML and pseudocode vs. pseudocode	condition synchronization
saUML/UML$_M$	saUML and source code vs. UML and source code	mutex locks only
saUML/UML$_C$	saUML and source code vs. UML and source code	condition synchronization

Table 2: *Objective studies performed*

In order to more impartially assess the benefits of using saUML sequence diagrams, we then conducted three objective user studies, which we refer to as saUML+, saUML/UML$_M$, and saUML/UML$_C$ (Table 2) [Xie et al. 2007(a); Xie et al. 2008]. The study saUML+ (Section 7.1) compared saUML plus textual materials (pseudocode) to textual materials (pseudocode) alone on a small multi-threaded program whose complexity arises from its use of *condition synchronization*, i.e., explicit thread signaling using the `wait` and `signal`/`broadcast` primitives. The data collected from the study showed a statistically significant benefit to the use of saUML in conjunction with pseudocode in comprehension tasks for programs that employ concurrency, but could not discern whether these benefits owed to our adornments or merely to the use of a diagrammatic notation to accompany the text [Xie et al. 2007(a)].

To understand whether the benefits we measured were due to saUML's specific extensions, we ran two

additional studies (Section 7.2). Both saUML/UML$_M$ and saUML/UML$_C$ compared saUML sequence diagrams to the standard UML sequence diagrams to judge the two notations' relative effectiveness in enhancing novices' understanding of concurrent programs. The programs, however, differed in their levels of synchronization complexity. Specifically, saUML/UML$_M$ compared the two notations when used to understand programs of low synchronization complexity, as judged by their use of only simple synchronization primitives, such as mutex locks. Here, a beneficial trend was observed, but it did not rise to the level of statistical significance. In contrast, saUML/UML$_C$ compared the two notations on similar tasks but with programs with more complex synchronization constructs. In this case, the programs used condition synchronization (e.g. wait and signal/broadcast). Here, a significant benefit was found to exist. Section 7.3 discusses threats to validity of the three objective studies.

7.1 First Objective Study: saUML+

We conducted an empirical study to examine the benefits of using saUML sequence diagrams in conjunction with source code in comprehension tasks for programs that employ concurrency. 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 pseudocode-only group with that of a pseudocode-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, a 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 (p=0.027 on a two-tailed, heteroscedastic t-test) to the use of our notation. Differences in time to complete were not significant. Details of the study can be found in [Xie et al. 2007(a)].

7.2 Remaining Studies: saUML vs Standard UML

Whereas the positive effects of diagrammatic over purely textual representations are well documented [Pancake 1994; Mayer 1997], one question left open by our first objective study, saUML+, was whether the positive effect of saUML diagrams owes to our extensions. Said another way, is there anything special about saUML diagrams or would the same benefits be observed from another, less feature-rich, graphical notation? The second and third objective studies, described in this section, investigated this question by comparing saUML with standard UML 2.0 sequence diagrams.

Both studies used the same basic experimental design (Section 7.2.1). However, they involved programs with different levels of synchronization complexity. In both studies, the treatment group was more successful than the control group at answering questions involving the behavior of multi-threaded programs. This effect was not statistically significant in the case of saUML/UML$_M$, which used a program that did not involve condition synchronization (Section 7.2.2). However, we found a statistically significant benefit to the use of the saUML extensions ($p < 0.05$) in the case of saUML/UML$_C$, which used a program containing condition synchronization (Section 7.2.3). An *a posteriori* power analysis using the effect size and sample variance observed in saUML/UML$_M$ indicates we would have needed 60-70 participants to observe an effect in this study, which involved only 24 participants. Thus, it is not yet clear whether the benefits observed in saUML/UML$_C$ extend to programs that do not involve condition synchronization.

7.2.1 Study Design

These two studies employed a between-subjects, pre-test/post-test study design. Students from two offerings of an undergraduate software-design course at Michigan State University were partitioned into two equivalent groups as measured by scores on a pre-test. One group, hereafter the *treatment group*, referred to the program's source code and to a collection of saUML diagrams depicting the interactions of interest to the particular question. The other group, hereafter the *control group*, referred to the code and to standard UML 2.0 sequence diagrams depicting the same phenomena.

Each test comprised both *comprehension-level* and *application-level* questions. In Bloom's Taxonomy, *comprehension* involves "the ability to grasp the meaning of material" and is demonstrated by restating

material from one form to another or by explaining or summarizing material [Bloom 1956]. We used comprehension-level questions to evaluate participants' understanding of the concurrency concepts presented in lectures. In contrast, *application* is demonstrated by applying the learned material to solve problems in new and concrete situations. We used application-level questions to compare the effectiveness of representations by asking participants to use these representations to reason about or predict the future behavior of scenarios of concurrent program execution. In each study, the control group used traditional UML sequence diagrams to answer these questions, and the treatment group used saUML diagrams to answer the same questions.

We used pre-test scores on both comprehension-level and application-level questions to evaluate the participants' prior knowledge of concurrency concepts and to divide the participants into equivalent treatment and control groups. These groups then attended parallel lectures on how to use either traditional UML sequence diagrams (control group) or saUML sequence diagrams (treatment group) in modeling behaviors of concurrent programs. We used post-test scores on the comprehension-level questions to detect any effects that might have arisen from participation in lectures conducted by different instructors. Because comprehension-level questions do not refer to specific scenarios, the independent variables (saUML vs. standard UML) should not directly impact a participant's performance on these questions. We used post-test scores on the application-level questions to evaluate the relative effectiveness of the two representations.

Students received extra credit for their participation in the studies. For both studies, course material presented prior to the study session covered standard UML notation and basic concurrency concepts. Students in saUML/UML$_C$ had been taught about condition synchronization, while participants in saUML/UML$_M$ had not. Thus, saUML/UML$_C$ was able to assess the use of the diagrams on more complex problems involving condition synchronization. Both of these studies involved two eighty-minute sessions. Each session began with a lecture and ended with a test. saUML/UML$_M$ was conducted with 24 students during the fall semester of 2006 and saUML/UML$_C$ was conducted with 38 students in the spring semester of 2007.

7.2.2 saUML/UML$_M$: No Condition Synchronization

We performed saUML/UML$_M$ to compare saUML with standard UML on problems with relatively simple synchronization complexity, i.e., threads and monitors with no condition synchronization. We conducted this study in two sessions. The first session began with a review of concurrency concepts followed by the pre-test. The second session began with a detailed introduction and a series of in-class demonstrations on the notation of interest (i.e., UML or saUML) followed by the post-test. All of the artifacts used in this study are available online at `http://www.cs.uga.edu/~eileen/SAUML_Studies`.

First Session.

In the first session of the study, all of the participants attended a lecture given by the instructor and then completed the pre-test. Results of the pre-test were used to partition the students into equivalent groups based on prior knowledge. During the lecture component, the researcher reviewed basic concurrency concepts and used standard UML sequence diagrams to demonstrate scenarios of interaction in systems where threads synchronize with one another using monitors. The running example involved two threads sharing access to a queue. The discussion involved pointing out several instances of data-access anomalies and also legal but often unexpected behaviors, focusing on how scenarios involving these anomalies/behaviors are depicted using sequence diagrams. The choice of anomalies and unexpected behaviors was informed by our earlier instructor survey of topics that students often miss or find difficult to envision [Xie et al. 2007(b)].

Fig. 3 depicts an example diagram, which we used to illustrate and probe student understanding of monitor semantics. Here, two threads—actor1 and actor2—attempt to pull an item off of a shared queue, and the threads are scheduled in such a way that their activations of the `pull` method overlap in time. The researcher would display such a diagram and then ask whether (1) the scenario is feasible in general and (2) it remains feasible if the queue is implemented as a monitor. In this instance, the answer to both questions is "yes."[7] Fig. 4 depicts a scenario that is feasible in general but not feasible if the queue is

[7]Students often fail to understand that a thread may invoke a method on a monitor, as depicted by the execution specification activated by actor2 in the middle of the activation by actor1. Of course, the former activation will immediately block until the monitor lock is released by actor1.

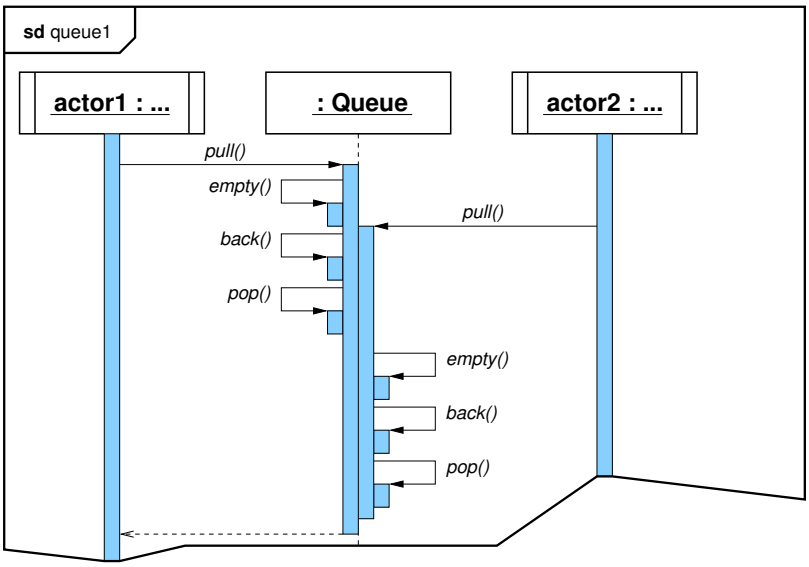


Figure 3: *One sample scenario used in probing students' understanding of monitors for saUML/UML$_M$.*

implemented as a monitor.

Following this lecture, the researcher then administered the pre-test. The pre-test contained nine application-level questions, each of which required participants to interpret the interactions between two concurrent threads in a specific execution scenario, and two comprehension-level questions, designed to gauge mastery of the notion of a "race condition" and knowledge of the major functions of a monitor. Each application-level question presented participants with a standard UML sequence diagram and up to five different candidate descriptions of the interaction depicted. Students were asked to select the candidate that best describes the depicted behavior. All of the scenarios were based on one of two different versions of a shared queue example mentioned previously. In one version, the shared queue is assumed to have been implemented with monitor semantics; whereas in the other version, the shared queue is implemented as an unprotected queue that can be accessed by any thread at any time. Fig. 5 lists one of the questions we asked on the pre-test. Notice that it refers to a specific diagram and asks what could happen next, i.e., how the scenario could play out following what is depicted in the diagram.

Based on pre-test scores, we divided the participants into a control group and a treatment group with equal means and standard deviations of score. The scores included answers to all multiple-choice questions, which included all of the application-level and one of the comprehension-level questions. The other

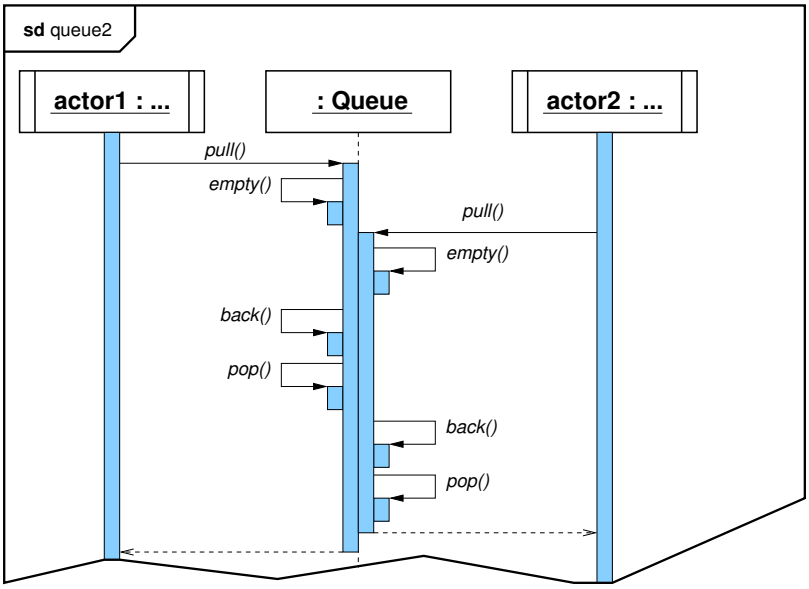


Figure 4: *A second sample scenario used in probing students' understanding of monitors for saUML/UML$_M$.*

___ Q1. Assume the *Queue* initially contains only Object A. What happens as a result of actors 1 and 2 executing the pull method as seen in Diagram 1?

a. actor1 gets a copy of Object A; actor2 gets nothing; the *Queue* becomes empty.
b. actor1 gets a copy of Object A; actor2 gets a copy of Object A; the *Queue* becomes corrupted.
c. actor1 gets a copy of Object A; actor2 gets a copy of Object A; the *Queue* becomes empty.
d. actor1 gets a copy of Object A; actor2 gets nothing; the *Queue* becomes corrupted.

Figure 5: *Sample application-level question for saUML/UML$_M$.*

comprehension-level questions was "open ended" and thus difficult to score accurately.

Table 3 summarizes the results of participants' scores on these pre-test questions, as well as the application-level questions alone, normalized to the range 0.0 to 1.0. Some drop-outs occurred between the pre-test and post-test sessions. Thus, the means of the two groups reported vary slightly, but not significantly, and with no detrimental effect on the our ability to perform subsequent analysis.

Second Session.

The second session consisted of a lecture and a post-test. The treatment and control groups attended parallel lectures, in two different classrooms. The two lectures reviewed the same concurrency and syn-

Table 3: *saUML/UML$_M$, mutex locks only, pre-test results, normalized*

	Application-level		All multiple-choice	
	Mean	STDV	Mean	STDV
Treatment	0.694	0.207	0.658	0.202
Control	0.741	0.191	0.683	0.170

chronization concepts and covered the same examples using the same textual descriptions. The two lectures differed in that the control group was presented with standard UML sequence diagrams, while the treatment group was presented with saUML diagrams.

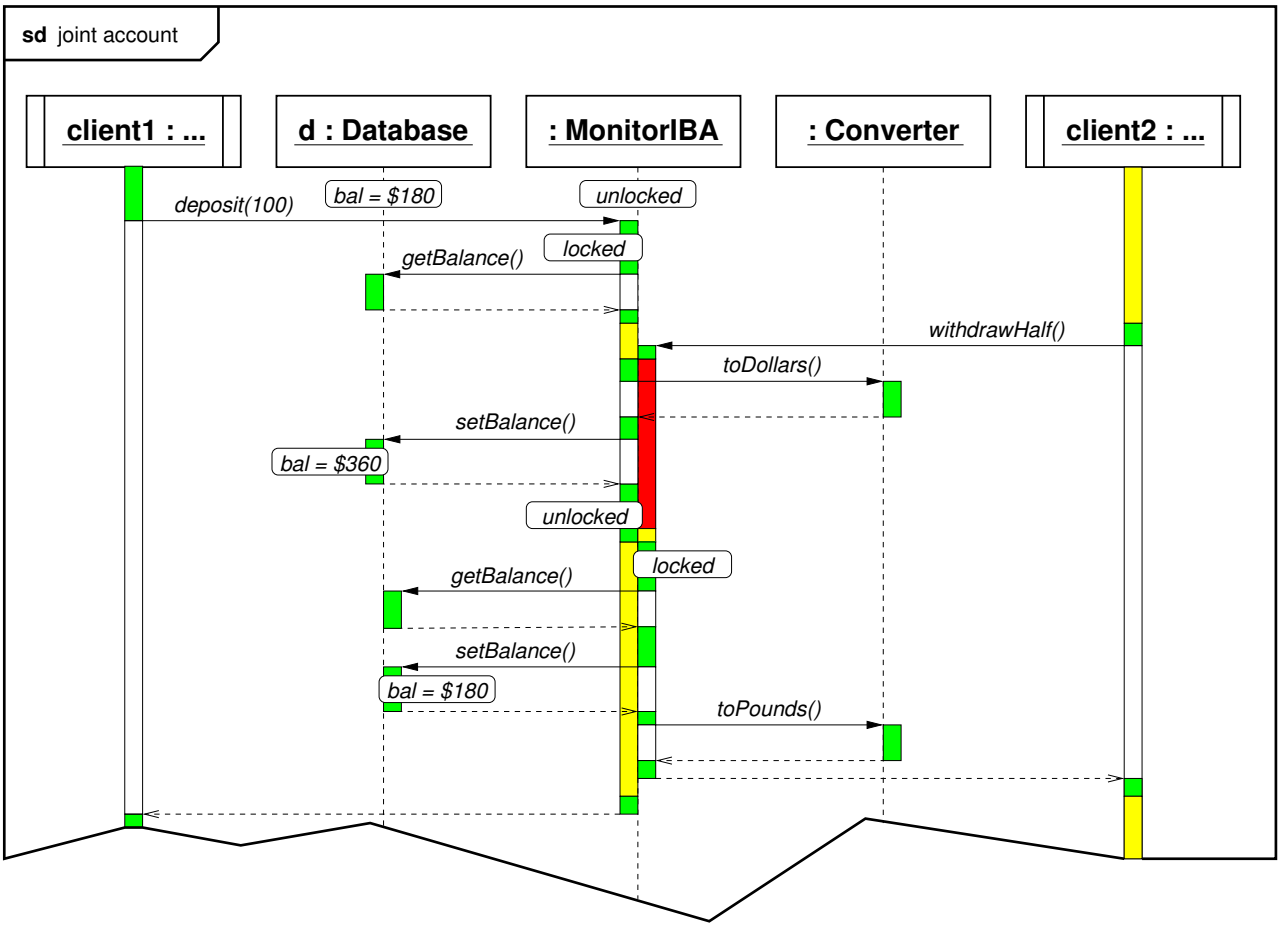


Figure 6: *A saUML diagram used in the post-test of saUML/UML$_M$.*

After the lecture, the groups were brought into one classroom to take the post-test, which included nine application-level questions and five comprehension-level questions. The application-level questions covered scenarios involving a monitor implementation of a shared international bank account upon which two concurrent client threads invoked deposit and withdrawal methods. These scenarios were presented to the

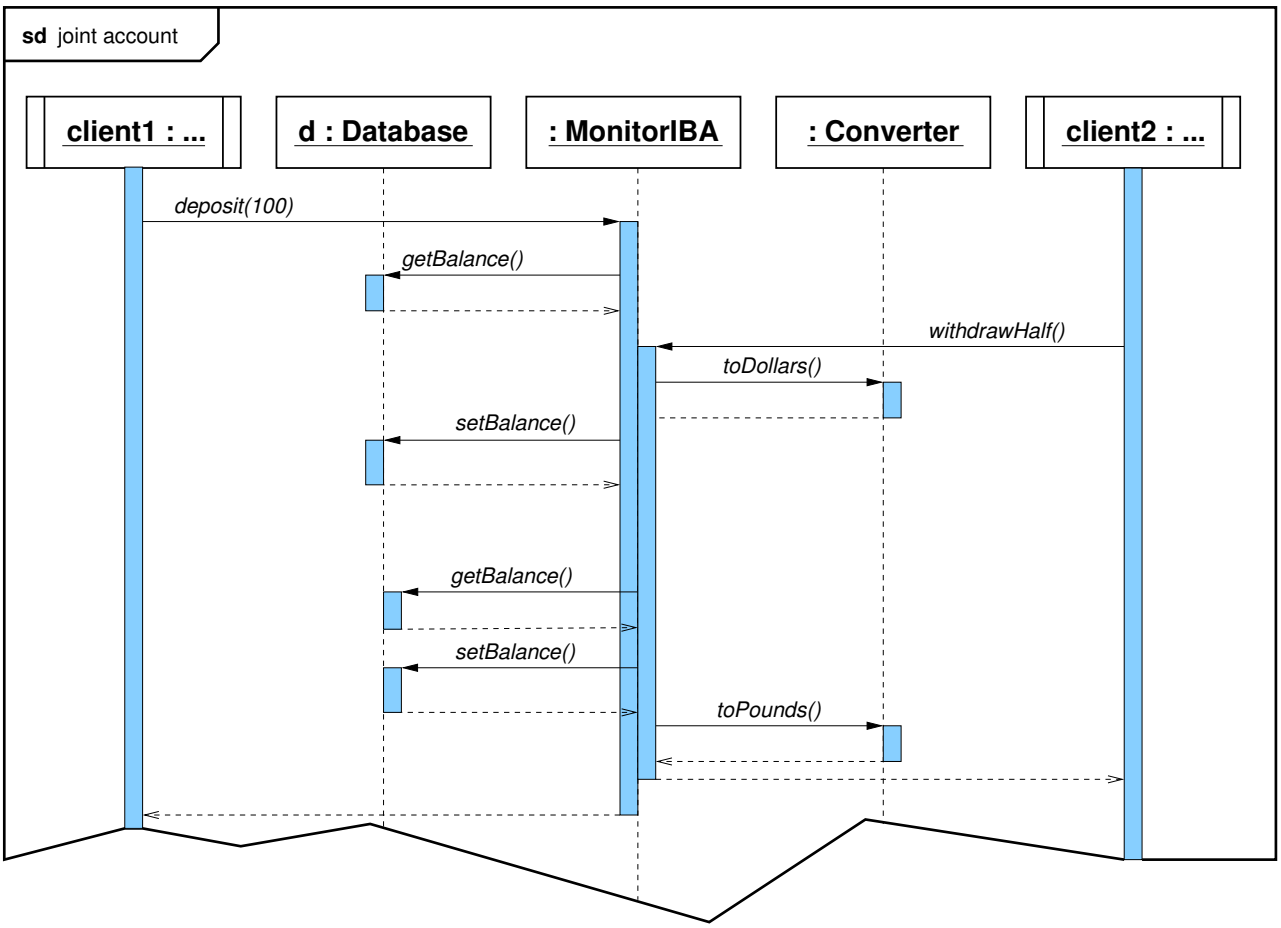


Figure 7: *A standard UML sequence diagram used in the post-test of saUML/UML$_M$, depicting the same scenario as depicted in Fig. 6.*

treatment group using saUML sequence diagrams and to the control group using standard UML sequence diagrams.

Appendix A lists the source code for class `IBA`, whose instances represent *international bank accounts*. These objects store balances in US Dollars but allow deposits and withdrawals in British Pounds. Class `MonitorIBA` extends class `IBA` by extending each of its operations into a monitor operation. Instances of this class are international bank accounts that execute as monitors.

Fig. 6 presents a sample saUML sequence diagram in which thread client2 invokes the monitor operation `withdrawHalf`. Because this invocation occurs while another thread (client1) is executing a monitor operation, `deposit(100)`, client2 suspends, waiting to enter the monitor. As indicated in the diagram, client2 remains suspended (red execution specification) until client1 unlocks the monitor, after

___ 8. What happens as a result of the execution depicted in Scenario 6?

a. withdrawHalf returns 50 GBP and the account now contains $90
b. withdrawHalf returns 100 GBP and the account now contains $270
c. withdrawHalf returns 50 GBP and the account now contains $270
d. withdrawHalf returns 100 GBP and the account now contains $180

Figure 8: *Sample question from the post-test of saUML/UML$_M$.*

which the thread state of client2 changes from suspended to ready (yellow). Notice that a context switch occurs shortly after client1 unlocks the monitor but before it can actually return from its invocation of `deposit(100)`. Because of this context switch, client2 was able to enter the monitor and complete his invocation of `withdrawHalf` before client1 is again scheduled to run and thus able to return. This example illustrates the kinds of unexpected timing phenomena that saUML diagrams clearly explain but that are more difficult to understand using only standard UML (Fig. 7). Fig. 8 presents a sample post-test question that targets the program execution scenario depicted in these diagrams.

Results.

Table 4: *saUML/UML$_M$, mutex locks only, post-test results, normalized*

	Application-level		Comprehension-level	
	Mean	STDV	Mean	STDV
Treatment	0.880	0.152	0.616	0.248
Control	0.806	0.171	0.666	0.246

Table 4 summarizes the post-test means and standard deviations of participants' scores on the application-level questions and the comprehension-level questions, normalized to the range 0.0 to 1.0. On the comprehension-level questions, the control group (mean: 0.666) slightly outperformed the treatment group (mean: 0.616). A two-tailed, unequal variance t-test shows that this difference is not statistically significant. We interpret this lack of a significant difference in performance between groups on the comprehension-level questions as support for a roughly equivalent benefit from the parallel lecture sessions. On the application-level questions, the treatment group (mean: 0.880) slightly outperformed the control group (mean: 0.806) despite the treatment group's lower mean score on the pre-test ($0.658 < 0.683$), and lower mean score on the comprehension-level questions ($0.616 < 0.666$). Again, the difference was not statistically significant. Analysis was performed using both parametric (t-test with assumption on unequal variance) and

non-parametric (Wilcoxon rank-sum) methods.

7.2.3 saUML/UML$_C$: Condition Synchronization

We conducted the last objective user study in two sessions, using essentially the same protocol as in saUML/UML$_M$. The first session consisted of a lecture attended by all participants followed by a pre-test. The second session began with a detailed introduction and a series of in-class demonstrations on the treatment or control notation followed by a post-test. In contrast to the prior study, saUML/UML$_C$ included a greater number of participants (38 vs. 24) and evaluated the diagrams in the context of more complex synchronization constructs (i.e., condition synchronization).

First Session. As in saUML/UML$_M$, the first session of saUML/UML$_C$ began with a lecture that covered thread synchronization, mutual exclusion, and monitor constructs. In addition, condition synchronization was reviewed. The pre-test materials of saUML/UML$_C$ were similar to those of saUML/UML$_M$ containing the same two comprehension-level questions, very similar application-level questions, and targeting the same shared-queue problem. Based on pre-test scores on the multiple-choice questions, we divided the participants into two equivalent groups.

Due to drop outs, the treatment group had 18 participants versus 20 in the control group. Table 5 summarizes the means and standard deviations of the participants' scores on these pre-test questions, as well as the application-level questions only, normalized to the range 0.0 to 1.0. The two groups had similar means and standard deviations of scores.

Table 5: *saUML/UML$_C$, condition synch, pre-test results, normalized*

	Application-level		All multiple-choice	
	Mean	St_dev	Mean	St_dev
Treatment	0.672	0.202	0.653	0.207
Control	0.650	0.207	0.625	0.217

Second Session. The second session consisted of a lecture and a post-test. As in saUML/UML$_M$, the treatment and control groups attended parallel lectures in different classrooms. Each lecture involved a review of the same concurrency and synchronization concepts, but diagrams used for the control group

were standard UML whereas diagrams used for the treatment group were saUML.

Following the lecture, the two groups were brought into the same classroom to take the post-test, which comprised three comprehension-level and eleven application-level questions. The application-level questions were based on execution scenarios involving a solution to the readers-writers problem [Lamport 1977], which involves condition synchronization and is more complex than the international bank account problem of saUML/UML$_M$. These scenarios were depicted using saUML diagrams for the treatment group and standard UML sequence diagrams for the control group.

Each scenario describes a collaboration between two threads, which are attempting to simultaneously access a shared database of account information. Threads in this study could play one of two distinct roles, *reader* or *writer*, where readers may access the database concurrently, but writer accesses must execute exclusive of any other reader or writer. Reader threads interact with the database by *bracketing* accesses with calls to two, potentially blocking, operations—`startRead` and `stopRead`. Writer threads bracket their accesses in a similar fashion using calls to `startWrite` and `stopWrite`. Condition synchronization is implemented by means of a mutex lock, two counter variables, `nReaders` and `nWriters`, and two condition-variable objects, `okToRead` and `okToWrite`.

The post-test questions refer to scenarios involving either two readers, one reader and one writer, or two writers. As an example, Fig. 9 depicts the UML and saUML diagrams representing a scenario in which a writer invokes `startWrite` after a reader has been granted read access to the database. Immediately following the invocation of `startRead`, the counter variable `nReaders` is non-zero; thus, when the writer invokes `startWrite`, it is able to acquire the mutex lock but must suspend itself until such time as both counter variables are 0. This is accomplished by invoking `wait` on the condition-variable object `okToWrite`.

Fig. 10 depicts another scenario involving reader-writer synchronization. Both diagrams indicate that the writer thread blocks waiting to acquire the mutex lock needed to enter the monitor. This property is clearly evident in the saUML diagram because the execution specification for `startWrite` turns red. The same property can be inferred from the UML diagram, because, had the writer acquired the lock first, `startWrite` would have returned prior to the return of `startRead`. Thus, while all of the information needed

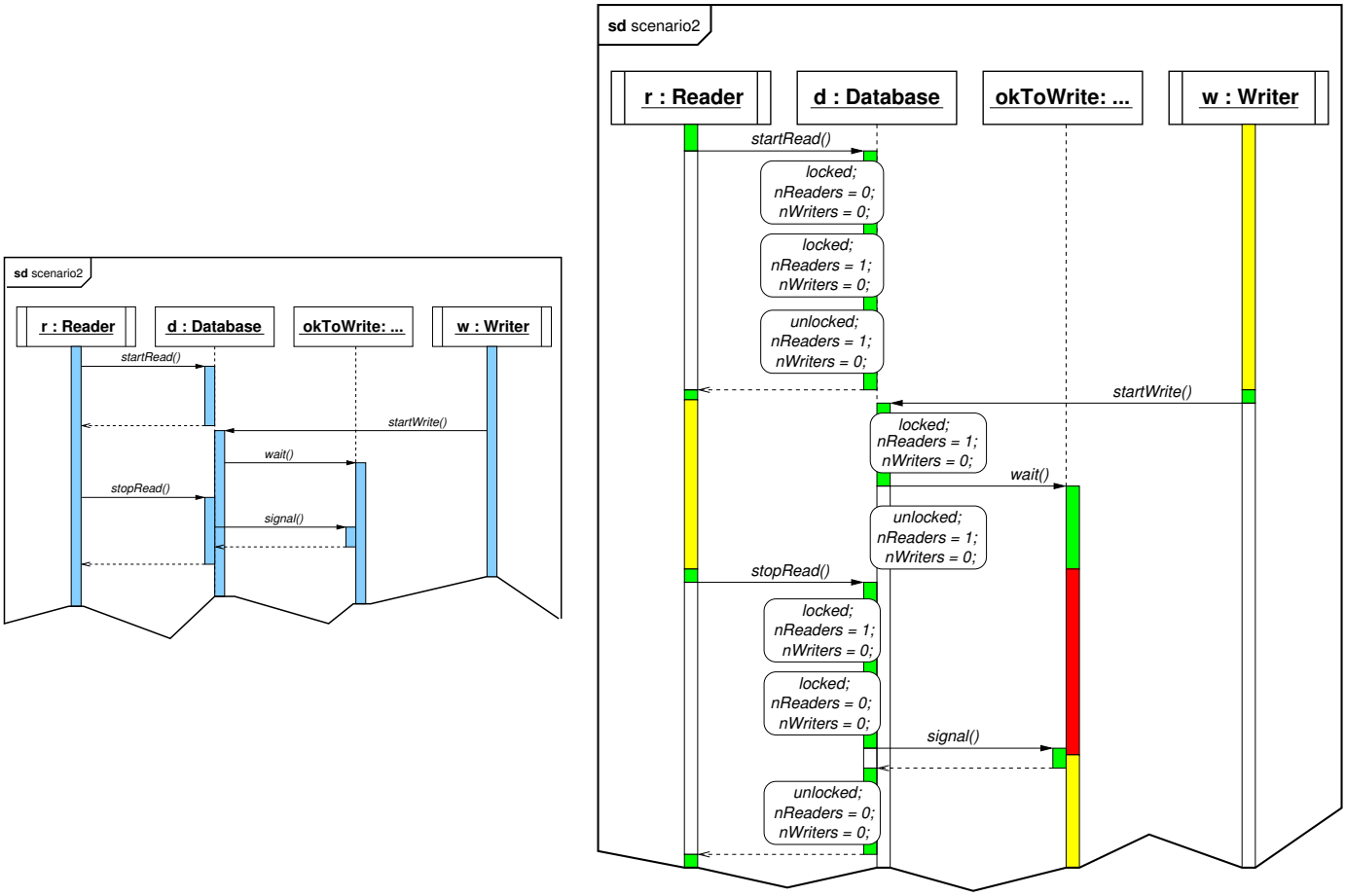


Figure 9: *UML and saUML renderings of a scenario pertaining to Question 3 on the post-test of saUML/UML$_C$.*

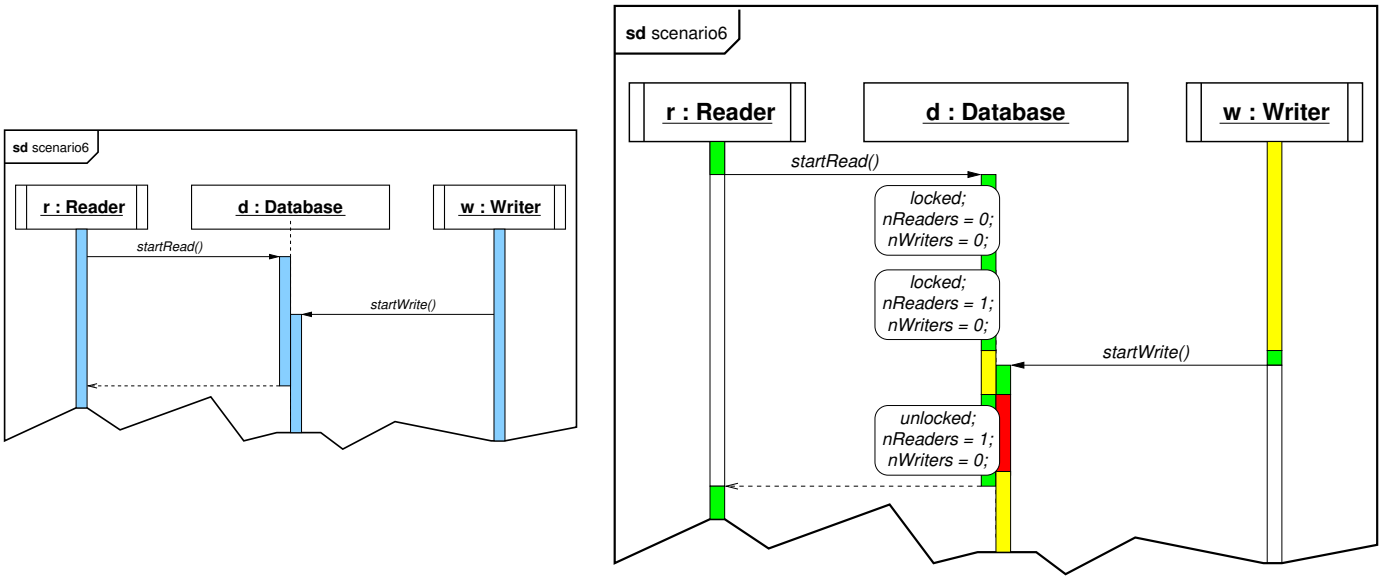


Figure 10: *UML and saUML renderings of a scenario pertaining to Question 7 on the post-test of saUML/UML$_C$.*

to check this property is "in" the standard UML representation, it is more clearly shown in the saUML representation.

Results.

Table 6 summarizes the results of the post-test. As expected, the participants of both the control group and the treatment group found the post-test of saUML/UML$_C$ to be more difficult than that of saUML/UML$_M$. As in saUML/UML$_M$, no significant difference was found for the comprehension-level questions. We view this as support for a roughly equivalent experience by the two groups in the parallel lecture sessions.

A one-tailed, heteroscedastic t-test (assumes unequal variance) of the score matrix of the post-test for saUML/UML$_C$ showed a statistically significant benefit to the use of the saUML diagrams ($p < 0.05$). These data were also analyzed with the Wilcoxon rank-sum test, a non-parametric alternative to the two-sample t-test. Again, a significant result was found ($p < 0.05$).

Table 6: *saUML/UML$_C$, condition synch, post-test results, normalized*

	Application-level		Comprehension-level	
	Mean	St_dev	Mean	St_dev
Treatment	0.642	0.237	0.593	0.25
Control	0.482	0.259	0.683	0.33

On a per-question basis, our data revealed a trend toward better performance using the saUML diagrams, particularly for the more difficult questions. For example, on the question associated with Fig. 9, only 20% of the UML users were able to answer correctly, while 39% of the saUML were able to do so. Similarly, only 25% of UML users answered the question associated with Fig. 10 correctly, while 50% of saUML users did so, a significant difference ($p < 0.05$).

That questions involving such scenarios would be difficult to answer is not surprising: Threads transition among several synchronization states and many operations are invoked in a short span of time. That said, both groups had access to the source code and to a textual description of the scenario, which included details such as that "Only one reader thread and one writer thread are running on the processor," that "The writer thread is in the suspended state (suspended on wait(OKtoWrite))," and that when the reader invokes `stopRead` it "sets numReaders to 0 and issues a notify(OKtoWrite)." That the participants using the

saUML diagram fared better on questions involving such a scenario suggests that the way in which the information is depicted and conveyed in the saUML diagram allows a larger number of participants to correctly reason about the behavior.

In summary, the saUML sequence diagram notation provides significant benefits over the standard UML sequence diagram notation when used by novices as an aid in answering application-level questions involving complex synchronization constructs (mutual exclusion, monitors, and condition synchronization).

7.3 Threats to Validity

Several threats to validity may be found in the design and conduct of the experiments comparing saUML and UML. First, these experiments were conducted with novices, students recently introduced to both concurrency concepts and these notations. Additional studies will be required to determine if the benefits seen to derive from saUML for these participants are also found with more experienced participants. Second, while the treatment and control groups were allocated identical amounts of time and overall group times were roughly the same (completing within one or two minutes of one another), individual completion times were not recorded for all studies (differences were not significant where recorded). In future studies, these times will be recorded and relevant analysis performed.

The study design involved the treatment and control groups attending parallel lectures during which they reviewed and gained some experience with the use of these notations to depict concurrency. Thus, the groups experienced these lectures with different instructors. Further, the experimenters were the instructors. We used the comprehension-level questions, which focused on general knowledge of concurrency, rather than problem-solving in a specific context, to gain insight into the impact of these factors, and found no difference in scores on these questions. This suggests that the two groups obtained a roughly equivalent benefit from attending the two different lectures. However, it is possible that a difference in benefit may have manifested only in the problem-solving, application-level questions. This threat could be reduced in future studies by using the same, non-experimenter lecturer to conduct these sessions. Finally, the style of question used to evaluate performance may fail to capture all relevant facets of understanding of concurrency. Future studies should expand the breadth of the questions to more broadly evaluate the

benefit.

8 Conclusion

Our long-term research 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 external representations can both aid students in mastering concurrency and synchronization concepts and potentially enable practitioners to better comprehend the dynamically evolving nature of concurrent programs. To this end, we have begun to investigate the common problems novices encounter in learning about concurrency and synchronization. We have also proposed saUML to address those problems and conducted empirical studies to assess the effectiveness of saUML sequence diagrams as aids to tasks that involve reasoning about the behavior of concurrent programs.

Our studies revealed that saUML sequence diagrams provide significant benefits over both textual representations and standard UML sequence diagrams for reasoning about concrete scenarios of program behavior that involve condition synchronization. Clearly, something about explicitly depicting thread states and synchronization mechanisms in the sequence-diagram format makes concurrent software easier for novice programmers to comprehend than when such information is left implicit. Moreover, our findings suggest that saUML may provide some benefit, although less pronounced, for reasoning about programs with relatively simple synchronization logic—that is, logic that involves mutexes but not condition variables. These results could inform the design of new tools and notations for visualizing concurrent software, especially software that uses condition synchronization. Whether such diagrams would be useful to practitioners is an open question.

Several interesting research questions remain regarding saUML. The most pressing concerns whether the benefits of saUML are limited to programs with complex condition synchronization. In our study saUML/UML$_M$, participants using saUML outperformed those using standard UML, but the difference

did not rise to statistical significance. This lack of statistical significance may be explained by the small sample size. An *a posteriori* power analysis shows that the statistical power of saUML/UML$_M$ was only 0.283, which means there is roughly a 70% chance we missed an effect. Assuming the effect size we observed holds, we would need a sample size of 65 to show statistical significance. We are currently working to replicate this study using participants from several universities to create a sufficiently large sample.

The saUML notation comprises several UML extensions and idioms of use. Further studies are needed to judge whether all of the extensions are needed or if a subset is sufficient. Moreover, having now used saUML in several studies, we have identified several optimizations that might improve its usability. For instance, complex synchronization states, such as that of the database in the second experiment, comprise many orthogonal components (e.g., state of the mutex lock and the value of each counter variable). Our current convention is to display the entire synchronization state (i.e., every component) when any one of them changes. Whether readability would improve if we depict only the components that change is an open question.

Our studies looked at tasks that involve reasoning about existing diagrams. Whether saUML is beneficial for tasks that involve creating diagrams from scratch is an open question. There are also questions regarding how well saUML scales for larger programs, especially compared with standard UML. For example, does saUML provide a significant benefit over standard UML on programs that use only mutexes if the programs are large, or utilize many, possibly nested, locks? Also, does saUML continue to provide a significant benefit for programs with condition synchronization if the programs are large or involve many condition variables?

We recognize that this research was conducted to improve educational benefit and that further study is required to determine whether and how it generalizes to practitioners. The programs and interaction scenarios used in our study may not be representative of those found in practice. Also, student participants may not be representative of expert practitioners, who have years of experience working on concurrent software. Finally, the questions we used may not be representative of the sorts of questions that arise in practice. We will address these issues in future work with case studies of professional programmers

conducting real maintenance tasks on production systems.

References

ARISHOLM, E., BRIAND, L. C., HOVE, S. E., AND LABICHE, Y. 2006. The impact of UML documentation on software maintenance: An experimental evaluation. *IEEE Transactions on Software Engineering 32*, 6, 365–381.

BEN-DAVID KOLIKANT, Y. 2004. Learning concurrency: evolution of students' understanding of synchronization. *Int. J. Hum.-Comput. Stud. 60*, 2, 243–268.

BLOOM, B. S. 1956. *Taxonomy of Educational Objectives, Handbook I: Cognitive Domain*. McKay, New York.

CARR, S., MAYO, J., AND SHENE, C.-K. 2003. ThreadMentor: a pedagogical tool for multithreaded programming. *J. Educ. Resour. Comput. 3*, 1, 1.

CHOI, S.-E., AND LEWIS, E. C. 2000. A study of common pitfalls in simple multi-threaded programs. In *Proc. 31st SIGCSE Tech. Symp. Comput. Sci. Educ. (SIGCSE 2000)*, ACM, New York, NY, USA, 325–329.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

HIGGINBOTHAM, C. W., AND MORELLI, R. 1991. A system for teaching concurrent programming. In *Proc. 22nd SIGCSE Tech. Symp. Comput. Sci. Educ. (SIGCSE 1991)*, ACM, New York, NY, USA, 309–316.

KRAMER, J. 2007. Is abstraction the key to computing? *Commun. ACM 50*, 4, 36–42.

KUTAR, M., BRITTON, C., AND BARKER, T. 2002. A comparison of empirical study and cognitive dimensions analysis in the evaluation of UML diagrams. In *Proc. 14th Psychology of Programming Interest Group*.

KUZNIARZ, L., STARON, M., AND WOHLIN, C. 2004. An empirical study on using stereotype to improve understanding of UML models. In *Proc. 12th IEEE International Workshop on Program Comprehension*, IEEE Computer Society, Los Alamitos, CA, USA, 14 – 23.

LAMPORT, L. 1977. Concurrent reading and writing. *Commun. ACM 20*, 11, 806–811.

LEROUX, H., RÉQUILÉ-ROMANCZUK, A., AND MINGINS, C. 2003. Jacot: a tool to dynamically visualise the execution of concurrent java programs. In *PPPJ '03: Proceedings of the 2nd international conference on Principles and practice of programming in Java*, Computer Science Press, Inc., New York, NY, USA, 201–206.

MAGEE, J., AND KRAMER, J. 2007. *Concurrency: State Models and Java Programs*, second ed. Wiley.

MAYER, R. E. 1997. Multimedia learning: Are we asking the right questions? *Educational Psychologist 32*, 1, 1–19.

MCDOWELL, C. E., AND HELMBOLD, D. P. 1989. Debugging concurrent programs. *ACM Comput. Surv. 21*, 4, 593–622.

MEHNER, K., AND WAGNER, A. 2000. Visualizing the synchronization of Java-threads with UML. In *Proc. 2000 IEEE Int. Symp. Visual Languages (VL 2000)*, IEEE Computer Society, Washington, DC, USA, 199.

MEHNER, K. 2002. JaVis: A UML-based visualization and debugging environment for concurrent Java programs. In *Software Visualization*, S. Diehl, Ed., vol. 2269 of *LNCS*. Springer, 163–175.

NEWMAN, E., GREENHOUSE, A., AND SCHERLIS, W. L. 2001. Annotation-based diagrams for shared-data concurrency. In *In Proceedings of the Workshop on Concurrency Issues in UML at UML 2001*. unpublished, available at http://wooddes.intranet.gr/uml2001/Home.htm.

OBER, I., AND STAN, I. 1999. On the concurrent object model of UML. In *Proc. 5th Int. Euro-Par Conf. Parallel Process. (Euro-Par 1999)*, Springer-Verlag, London, UK, 1377–1384.

PANCAKE, C. M. 1994. Visualization techniques for parallel debugging and performance tuning tools. Tech. rep., Corvallis, OR, USA.

PURCHASE, H. C., MCGILL, M., COLPOYS, L., AND CARRINGTON, D. 2001. Graph drawing aesthetics and the comprehension of UML class diagrams: An empirical study. In *Proceedings, 2001 Asia-Pacific Symposium on Information Visualization*, vol. 9, 129–137.

RUMBAUGH, J., JACOBSON, I., AND BOOCH, G. 2004. *The Unified Modeling Language Reference Manual*, second ed. Addison–Wesley.

SCHADER, M., AND KORTHAUS, A. 1998. Modeling Java threads in UML. In *The Unified Modeling Language – Technical Aspects and Applications*, Physica-Verlag, Heidelberg, M. Schader and A. Korthaus, Eds., 122–143.

SHENE, C.-K., AND CARR, S. 1998. The design of a multithreaded programming course and its accompanying. *Software Tools, The Journal of Computing in Small Colleges 14*, 12–24.

STEVENS, P. 2003. UML and concurrency. In *Abstract State Machines*, 151–165.

SWAN, J., BARKER, T., BRITTON, C., AND KUTAR, M. 2005. An empirical study of factors that affect user performance when using uml interaction diagrams. In *Proceedings, 2005 International Symposium on Empirical Software Engineering*, 10.

TILLEY, S., AND HUANG, S. 2003. A qualitative assessment of the efficacy of UML diagrams as a form of graphical documentation in aiding program understanding. In *Proceedings of the 21st Annual International Conference on Documentation*, 184 – 191.

TORCHIANO, M. 2004. Empirical assessment of UML static object diagrams. In *Proc. 12th IEEE International Workshop on Program Comprehension*, IEEE Computer Society, Los Alamitos, CA, USA, 226 – 230.

XIE, S., KRAEMER, E., AND STIREWALT, R. E. K. 2007(a). Empirical evaluation of a UML sequence diagram with adornments to support understanding of thread interactions. In *Proc. 15th IEEE Int. Conf. Program Comprehension (ICPC 2007)*, IEEE Computer Society, Washington, DC, USA, 123–134.

XIE, S., KRAEMER, E., AND STIREWALT, R. E. K. 2007(b). Design and evaluation of a diagrammatic notation to aid in the understanding of concurrency concepts. In *Proc. 29th Int. Conf. Software Eng.*

(ICSE 2007), IEEE Computer Society, Washington, DC, USA, 727–731.

XIE, S., KRAEMER, E., STIREWALT, R. E. K., DILLON, L. K., AND FLEMING, S. D. 2008. Assessing the benefits of synchronization-adorned sequence diagrams: two controlled experiments. In *SoftVis '08: Proceedings of the 4th ACM symposium on Software visuallization*, ACM, New York, NY, USA, 9–18.

A Appendix

```
class IBA {
 public:
  ...
  void deposit(double amount) // amount in GBP
  {double balance(db->getBalance(acctID);
   balance += currencyConv->toDollars(amount);
   db->setBalance(acctID, balance);
  }
  double withdrawHalf() // amount in GBP
  {double balance(db->getBalance(acctID);
   balance /= 2.0;
   db->setBalance(acctID, balance);
   return currencyConv->toPounds(balance);
  }
 private:
  unsigned acctID; // customer acct #
  Database* db; // acct balances in USD
  Converter* currencyConv; // converts USD to/from GBP
};
class MonitorIBA : public IBA {
 public:
  ...
  void deposit(double amount) // amount in GBP
  {pthread_mutex_lock(&lock);
   IBA::deposit(amount);
   pthread_mutex_unlock(&lock);
  }
  double withdrawHalf() // amount in GBP
```

```
  {pthread_mutex_lock(&lock);
   double amount=IBA::withdrawHalf();
   pthread_mutex_unlock(&lock);
   return amount;
  }
 private:
  pthread_mutex_t lock;
};
```

A monitor implementation of a shared bank account.