Assessing the Benefits of Synchronization-Adorned Sequence Diagrams: Two Controlled Experiments*

Shaohua Xie, Eileen Kraemer
Department of Computer Science
University of Georgia
Athens, Georgia, USA 30602
{shaohua,eileen}@cs.uga.edu

R. E. K. Stirewalt, Laura K. Dillon, Scott D. Fleming
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan, USA 48824
{stire,ldillon,sdf}@cse.msu.edu

Abstract

Learning about concurrency and synchronization is difficult for novices. In prior work, we developed *saUML*, a refinement of UML sequence diagrams, to address these difficulties and found them to be beneficial when compared to text-only presentations. This paper compares saUML to standard UML sequence diagrams to judge their relative effectiveness in enhancing a novice programmer's understanding of programs with different levels of synchronization complexity. One experiment 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. A second experiment compared the two notations on similar tasks but on programs with more complex synchronization constructs, in this case condition synchronization using primitives, such as wait and signal. Here, a significant benefit ($p < 0.05$) was found to exist.

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; 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 using low-level primitives, which synchronize threads indirectly by modifying OS-level data structures such as condition queues and mutex locks.

In prior work, we reported how *synchronization-adorned UML* (saUML) sequence diagrams aid novice programmers in understanding tasks when compared with purely textual representations [Xie et al. 2007b]. Whereas the positive effects of diagrammatic over purely textual representations are well documented [Pancake 1994], one question left open by our prior work 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? Conceivably, the added adornments might actually detract from understanding by producing busier diagrams. This paper investigates this question by comparing saUML with a simpler notation—standard UML 2.0 sequence diagrams.

Using a combination of color and textual adornments, saUML extends UML 2.0 sequence diagrams to depict the run-time states of threads, mutex locks, and counter variables. We chose to depict this information based on responses to an instructor survey, which elicited the concepts and issues that students find most difficult when learning about concurrency [Xie et al. 2007a]. By making these concepts explicit, saUML diagrams expose many of the otherwise invisible details in play during thread synchronization. In a previous study [Xie et al. 2007b], we compared saUML plus textual materials to textual materials alone on a small multithreaded program whose complexity arises from its use of *condition synchronization*, i.e., explicit thread signaling using the `wait` and `signal`/`broadcast` primitives. This study revealed statistically significant benefits to the use of saUML but could not discern whether these benefits owed to our extensions or merely to the use of a diagrammatic notation to accompany the text.

To understand whether the benefits we measured in the previous study were due to saUML's specific extensions, we ran two user studies comparing saUML to standard UML. The first study involved programs that use relatively simple synchronization, whereas the second involved programs that use condition synchronization. The participants in both studies were computer science students enrolled in two different offerings of a junior-level software engineering course. For each study, students 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.

*The material is based upon work supported by the National Science Foundation under Grants CCF-0702667 and IIS-0308063. Additional support provided by ONR grant N00014-01-1-0744 and by LogicBlox, Inc. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Office of Naval Research, or LogicBlox, Inc.

In both studies, the treatment group was more successful than the control group at answering questions involving the behavior of multi-threaded programs. The second study, which involved programs that use condition synchronization, demonstrated a statistically significant benefit to the use of the saUML extensions ($p < 0.05$). It is not yet clear whether the benefit extends to programs involving only simple synchronization. An *a posteriori* power analysis using the effect size and sample variance we observed in our first study indicates we would have needed 60-70 participants to observe an effect, and our study involved only 24 participants.

The remainder of the paper is structured as follows. By way of background, we introduce the saUML conventions and briefly survey the related work in graphical representations to support reasoning about concurrency and synchronization (Section 2). We then describe our overall experimental methods and materials (Section 3) and discuss the details and results of each experiment (Section 4, Section 5). Whether there exists a threshold of synchronization complexity below which saUML is no more effective than standard UML is an open question that will need to be explored in a larger study. Also, whether the beneficial effects of saUML scale to larger programs is an open question. We conclude with a discussion of new questions and future work (Section 6).

2 Background

We designed the saUML notation to extend UML sequence diagrams with features to address the difficulties commonly experienced by novices in learning about concurrency and synchronization [Xie et al. 2007a].

2.1 UML sequence diagrams

We focused on UML sequence diagrams for several reasons. First, analysts often construct concrete scenarios of interaction when diagnosing faults in multi-threaded programs [Fleming et al. 2008]. Of the many different notations for behavioral modeling in UML, the sequence and communication diagrams are best suited for depicting such scenarios. Swan and colleagues found that sequence diagrams are easier to learn than are communication diagrams and found significant benefit of one over the other among participants who were familiar with both [Swan et al. 2005].[1] In addition, we opted for sequence diagrams because object lifelines and activations are easily adorned with state information, which is awkward to depict in a communication diagram. We are exploring the use of the state-modeling notation in a study that is currently underway. Moreover, while a given sequence diagram depicts only a single scenario, a small collection of distinct diagrams often suffices to explain the essence of a concurrent design. Lastly, UML is commonly used to model object-oriented systems; in particular, the sequence diagram has been widely adopted in practice.

The sequence diagram provides several features that are useful in modeling interactions among concurrent agents—e.g., the ability to designate *active objects* and provisions for asynchronous message passing among active objects. By modeling each thread as an active object, a diagram visualizes time ordering of interactions between threads in a single program trace. Despite several attempts to model thread-level synchronization in earlier versions of UML sequence diagrams [Schader and Korthaus 1998; Mehner and Wagner 2000], the current notation provides little support for modeling these issues [Ober and Stan 1999; Stevens 2003].[2]

[1] The collaboration diagrams of UML 1.x were renamed *communication diagrams* in UML 2.0.

[2] Note: While 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].

2.2 saUML extensions

We designed saUML to support the design and understanding of 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.[3] In the sequel, 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, saUML diagrams depict a scenario of interaction among a collection of threads and monitors in this sense.

In designing saUML, we chose to depict the states of operating system resources (e.g., thread and lock states), application synchronization conditions, and details regarding how invocations of primitives affect those states and conditions. When assigning features to visual representations, we opted for combinations that support "at a glance" detection of global synchronization properties (e.g. deadlocks, safety violations). 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 (reader and writer) attempt to access a shared database (Fig. 1).

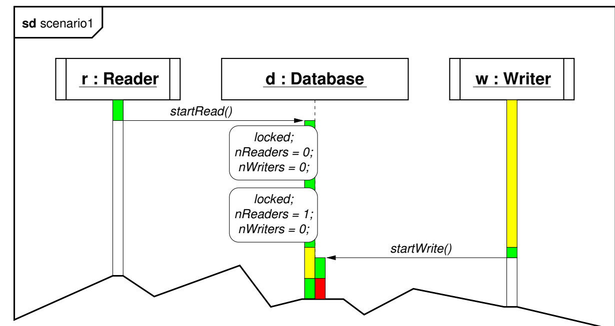


Figure 1: *Sample saUML sequence diagram.*

saUML extensions include two new features to represent the scheduling states of threads and the synchronization states of monitors. Active objects denote threads; they are designated using UML's double bar convention on the object box. Fig. 1 depicts a scenario involving two threads—a reader thread r and a writer thread w. Activation bars, called *execution specifications* in UML 2.0, are colored to indicate whether the executing thread is suspended (red), ready (yellow), or running (green). By convention, the thread is running if its most deeply nested execution specification[4] is shaded green, ready if this specification is shaded yellow, and suspended if it is shaded red.[5] Moreover, at any point in time, only the most deeply nested execution specification is shaded. In

[3] For instance, the database object in the readers–writer example is not a strict monitor because it allows concurrent reads.

[4] the activation at the top of the thread's run-time stack

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

Fig. 1, 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 lifeline states [Rumbaugh et al. 2004, pg 589], i.e, rounded rectangles containing assertions that describe the synchronization state. For instance, in Fig. 1, 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` 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. 1 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 suspends and the reader thread resumes. The rest of the scenario is omitted for brevity.

2.3 Related work

Our saUML extensions are most closely related to the sequence-diagram extensions proposed by Mehner and Wagner [Mehner and Wagner 2000]. They color execution specifications to distinguish when an activation is active (dark) or suspended (white). Unlike saUML, these conventions conflate 1) distinct thread states *ready* and *running*, both of which are depicted as active, and 2) thread blocking due to synchronization with the activation of a nested procedure, both of which are depicted as suspended. These fine distinctions made by saUML address many of the knowledge gaps upon which novices often stumble. Mehner and Wagner also code `wait` and `notify` operations as primitives of the monitor. Such a coding is consistent with Java's model of synchronization, but it does not easily scale to represent monitors with multiple condition variables. SaUML addresses this issue by modeling condition variables as distinct objects.

Several researchers have developed visualization tools to assist students in learning about concurrency and synchronization [Carr et al. 2003; Higginbotham and Morelli 1991; Leroux and Exton 2001]. One tool in particular, called *Jacot*, uses UML to depict thread interactions in Java programs [Leroux et al. 2003]. Unfortunately, these visualizations depict thread synchronization at a level that abstracts away many of the subtle details that novices find difficult to learn. For instance, when a thread invokes `wait` on a condition variable, the thread releases the mutex lock associated with that condition variable before suspending. However, because the lock is released inside the body of the `wait` primitive, novices often believe the thread that invoked `wait` will continue to hold the lock while waiting. By abstracting away such implementation details, the afore-mentioned visualizations may fail to expose this misconception.

A number of researchers have studied the usability of different UML diagrams. Kutar and colleagues compared the impact of collaboration and sequence diagrams on comprehension and found no significant difference in performance [Kutar et al. 2002]. In a later study, Swan and colleagues 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 [Swan et al. 2005]. Torchiano compared the use of class diagrams alone to the use of class diagrams plus object diagrams as aids in answering questions about simple programs [Torchiano 2004]. The study revealed that object diagrams provided benefits in some cases and were neutral in others.

Tilley and Huang investigated the efficacy of UML diagrams in aiding program understanding [Tilley and Huang 2003]. UML experts were given a series of UML diagrams and asked to answer questions about the depicted software 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 limiting 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 and effort of 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.

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

3 Study Design

We conducted two experiments to compare the effectiveness of saUML diagrams with that of standard UML sequence diagrams when used by novices in the comprehension of concurrent systems. The experiments employed a between-subjects, pre-test/post-test study design using students from two offerings of an undergraduate software-design course at Michigan State University. 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 translating material from one form to another, by explaining or summarizing material, or by predicting consequences or effects [Bloom 1956]. We used comprehension-level questions to evaluate participants' understanding of the concurrency concepts presented in the lectures. By contrast, application-level questions ask students to use learned material in new and concrete situations. We used application-level questions to compare the effectiveness of representations by asking participants to answer questions about scenarios of concurrent program execution. In each experiment, the *control group* used traditional UML sequence diagrams to answer these scenario 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 behavioral modeling of concurrent programs using either traditional UML sequence diagrams (control group) or saUML sequence diagrams (treatment group). 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.

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

4 First Experiment

We performed the first experiment 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 experiment in two sessions. The first session began with a review of concurrency concepts followed by the pre-test (Section 4.1). 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 (Section 4.2). All of the artifacts used in this study are available online at `http://www.cs.uga.edu/˜eileen/SAUML_Studies`.

4.1 First Session

In the first session of the experiment, all of the participants attended a lecture given by the experimenter 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 experimenter 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. 2007a].

Fig. 2 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 experimenter 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."[6] Fig. 3 depicts a scenario that is feasible in general but not

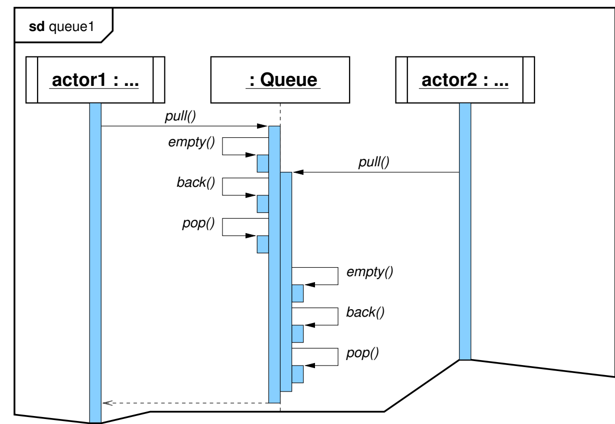


Figure 2: *One sample scenario used in probing students' understanding of monitors.*

feasible if the queue is implemented as a monitor.

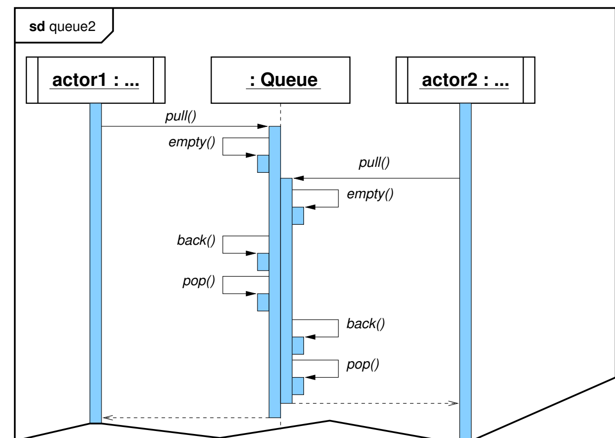


Figure 3: *A second sample scenario used in probing students' understanding of monitors.*

Following this lecture, the experimenter 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. 4 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

[6]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.

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 4: *Sample application-level question.*

questions, which included all of the application-level and one of the comprehension-level questions. The other comprehension-level questions was "open ended" and thus difficult to score accurately.

Table 1 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 with no detrimental effect on the our ability to perform subsequent analysis.

Table 1: *Experiment 1 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

4.2 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 synchronization 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.

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

Fig. 5 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)`[7], client2 suspends, waiting to enter the monitor. As indicated in the diagram, client2 remains suspended (red activation bar) until client1 unlocks the monitor, after 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.

```
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;
};
```

Figure 5: *A monitor implementation of a shared bank account.*

4.3 Results and Analysis

Table 2: *Experiment 1 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 2 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). This difference is not statistically significant. We interpret this lack of a significant difference in performance between groups on the

[7] Recall that the `IBA::deposit` takes its argument in British Pounds rather than US Dollars. This deposit is converted internally into 180 US Dollars. At the time we ran this study, the exchange rate was much more favorable to the Dollar than it is now.

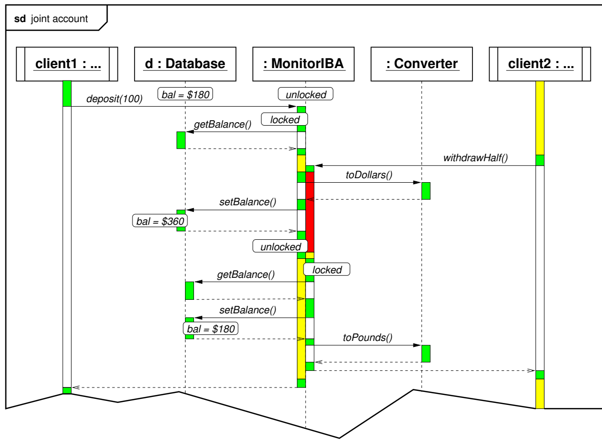


Figure 6: *A saUML diagram used in the post-test of the first experiment.*

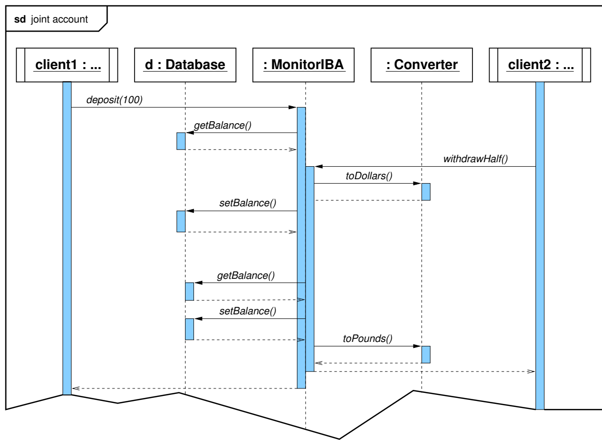


Figure 7: *A standard UML sequence diagram used in the post-test of the first experiment, depicting the same scenario as depicted in Fig. 6.*

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.

5 Second Experiment

We conducted the second experiment in two sessions, using essentially the same protocol as in the first experiment. The first session consisted of a lecture attended by all participants followed by a pre-test (Section 5.1). 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 (Section 5.2). In contrast to the first experiment, this one 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).

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 the first experiment.*

5.1 First Session

As in the first experiment, the first session of this experiment began with a lecture that covered thread synchronization, mutual exclusion, and monitor constructs. In addition, condition synchronization was reviewed. The pre-test materials of the second experiment were similar to those of the first experiment, 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 3 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 3: *Experiment 2, pre-test results, normalized*

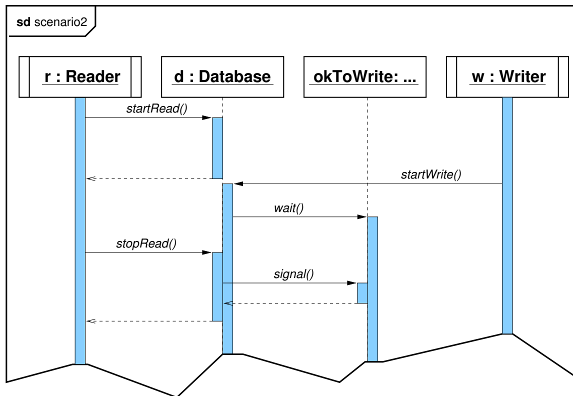
	Application-level		All multiple-choice	
	Mean	St_dev	Mean	St_dev
Treatment	0.672	0.202	0.653	0.207
Control	0.650	0.207	0.625	0.217

5.2 Second Session

The second session consisted of a lecture and a post-test. As in the first experiment, 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- 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 the first study. 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 experiment 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 `nRead-`



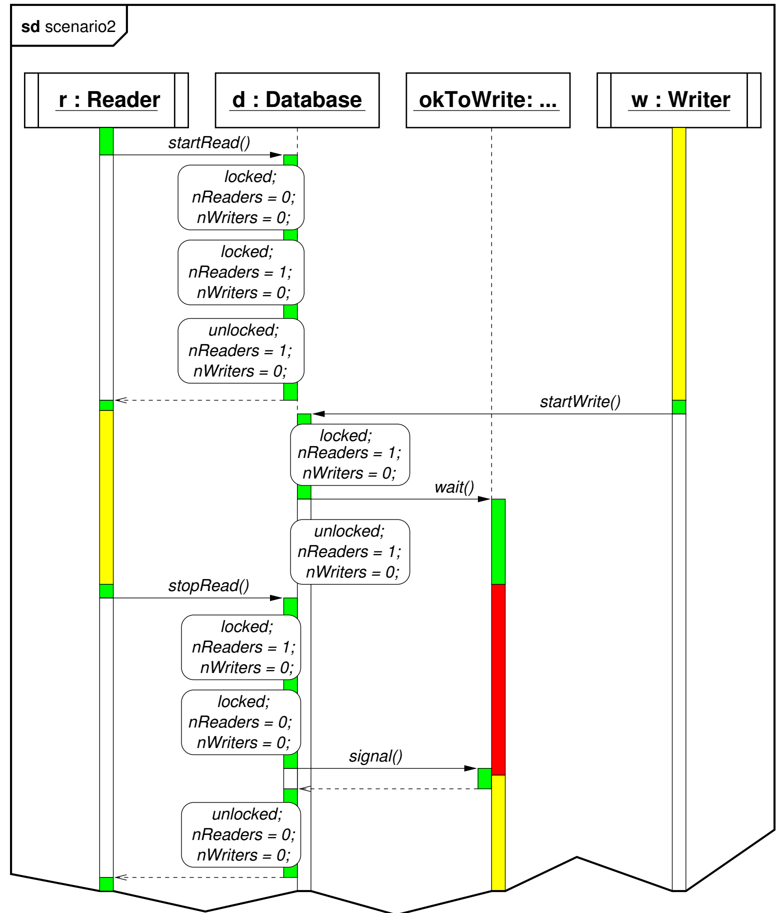


Figure 9: *UML and saUML renderings of a scenario pertaining to Question 3 on the post-test.*

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

5.3 Results and Analysis

Table 4 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 this second experiment to be more difficult than the first. As in the first experiment, 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 this second experiment 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 4: *Experiment 2, 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).

6 Conclusion

Our studies revealed that saUML extensions provide significant benefits over 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. Further, it stands to reason that if saUML helps students to learn about concurrent programming, it could potentially help practitioners with program-comprehension tasks.

Several threats to validity may be found in the design and conduct of these experiments. 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 in 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. 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.

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 first study, 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

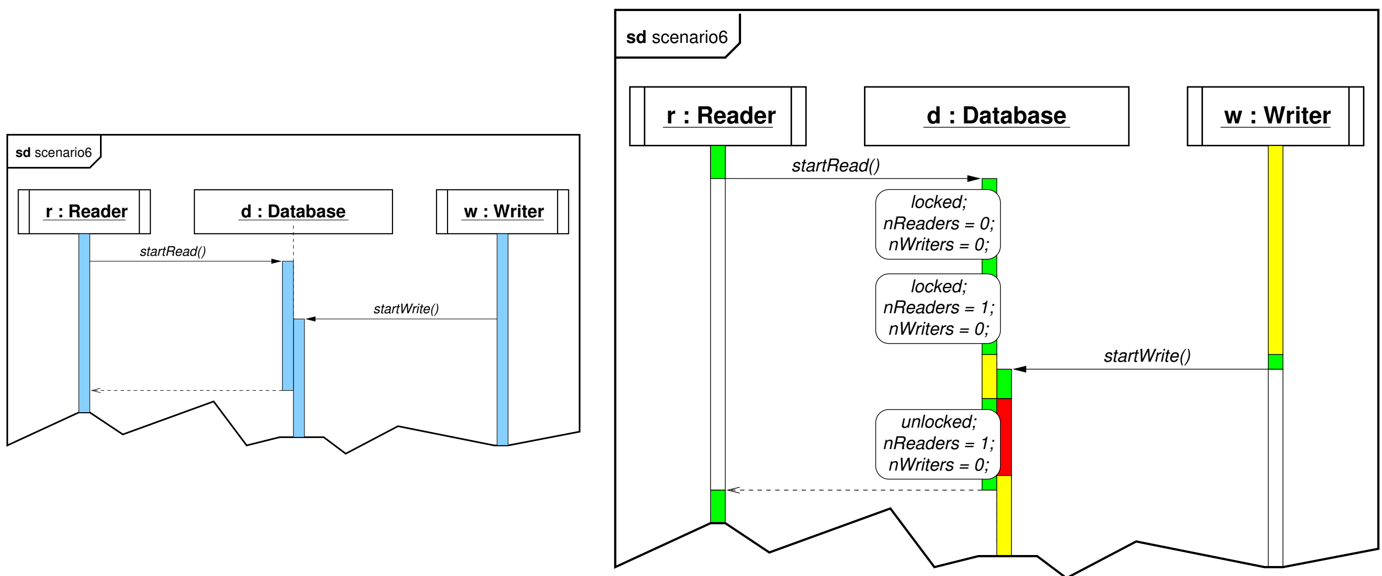


Figure 10: *UML and saUML renderings of a scenario pertaining to Question 7 on the post-test.*

the small sample size. An *a posteriori* power analysis shows that the statistical power of the first study 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.

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.

FLEMING, S. D., ET AL. 2008. A study of student strategies for the corrective maintenance of concurrent software. In *Proc. IEEE/ACM Int. Conf. Software Eng. (ICSE 2008)*.

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.

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

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., AND EXTON, C. 2001. Visualising the execution of concurrent object-oriented programs dynamically using UML. In *Proc. 9th Int. Conf. Central Europe Comput. Graph., Visualization and Comput. Vision (WSCG 2001)*.

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.

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.

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.

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. 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., AND STIREWALT, R. E. K. 2007. 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.